# 9

# Protokolování a ladění

V každé aplikaci, již vytváříte, mohou robustní mechanizmy protokolování a ladění uspořit hodiny úsilí a snah o sledování problémů, ke kterým může dojít.

Protokoly mohou pomoci při analýze historie užívání, kterou není možné určit z protokolů webového serveru – např. zátěž serveru, provedené dotazy SQL nebo specifické zprávy aplikace, jež by se jinak nezaznamenaly.

Laděním můžete zkoumat hodnoty proměnných, sledovat které podmíněné bloky se provedly nebo jinak pozorovat stav aplikace za běhu, aniž byste museli zaplnit kód příkazy `print`, jež je třeba před nasazením odstranit. Poté, co zpracujete kód v této kapitole, získáte množinu tříd, které by měly být součástí každého vašeho dalšího projektu.

## Vytvoření protokolovacího mechanizmu

Primárním účelem protokolování je umožnit zpětné sledování toho, co aplikace dělala a jak postupovala. Správně sestavený protokol vám umožní zpětně provést analýzu chování systému za účelem zjištění toho, zda nejsou zapotřebí změny.

### Jednoduché protokolování

Nejjednodušším způsobem, jak realizovat protokolování, je zapisovat informace do souboru. Uživatelský účet, pod kterým běží daný proces webového serveru, však zpravidla nemá potřebná oprávnění pro zápis do souborového systému, s výjimkou dočasného adresáře (`/tmp` na většině unixových systémů). To není úplně nejvhodnější umístění, kde skladovat důležité protokoly aplikace. Abyste umožnili procesu webového serveru zápis do souboru, vytvořte dedikovaný adresář, do kterého se budou ukládat protokoly:

Vytvořte adresář s názvem `logs`, nacházející se v hierarchii nad kořenovým adresářem vaší aplikace a ujistěte se, že uživatelský účet používaný webovým serverem má oprávnění k tomuto adresáři (v unixových systémech použijte příkazy `chmod 744` a `chown nobody`, popř. dosaďte

jméno uživatele. Uživatelé systému Windows najdou potřebné informace v dokumentaci své verze systému).

```
/www
|
|
+---./mysite
|
|
+---htdocs
|   |
|   |
+   +---images
|   |
|   +---css
|
+---logs
```

## Ukázková struktura souborového systému

Abyste v PHP mohli zapisovat do souboru, je třeba, pomocí funkce `fopen`, nejdříve získat handle tohoto souboru a specifikovat, že se má soubor otevřít pro zápis. Samotný zápis do souboru realizuje funkce `fwrite`. Funkce `fclose` následně zajistí zavření souboru. Následující kód definuje a poté i demonstruje funkci `logMessage`, která používá tyto základní funkce pro zápis do předdefinovaného protokolu:

```
<?php

function logMessage($message) {
  $LOGDIR = '/www/mysite/logs/'; //chmod 744 a chown nobody
  $logFile = $LOGDIR. 'mysite.log';
  $hFile = fopen($logFile, 'a+'); //otevření pro zápis na konec souboru, pokud
                                  //soubor neexistuje, vytvoří se

  if(!is_resource($hFile)) {
    printf("Soubor %s nelze otevřít pro zápis. Ověřte přístupová práva.",
      $logFile);
    return false;
  }

  fwrite($hFile, $message);
  fclose($hFile);
  return true;
}

logMessage("Ahoj světe!\n");
?>
```

Pokud tento kód spustíte a adresář uložený v proměnné `LOGDIR` má správně nastavená práva, měla by vás v souboru `mysite.log` očekávat zpráva `Ahoj světe!`. Pro úplně nejzákladnější potřeby bude tato funkce dostačovat, ale ve skutečné aplikaci budete chtít protokolovat více než pouhé zprávy. Protokol by měl být strojově zpracovatelným souborem, tj. data protokolu by měla být schopna zpracovat samostatná aplikace analyzující uložené informace. Protokol by měl obsahovat datum a čas každé zprávy, míru vážnosti nebo důležitosti zprávy a určitou zmínku o tom, který modul aplikace zprávu vygeneroval.

## Třída Logger

Třída z následující ukázky kódu nabízí OO mechanizmus pro generování detailnějšího protokolu než je ten z předchozí části kapitoly. Vytvořte soubor s názvem `class.Logger.php`. Tento soubor bude obsahovat definici třídy `Logger`, která umožňuje zápis informací do souboru na serveru ve formátu odděleném tabulátory. Ukládá se časové razítko, úroveň (vážnost zprávy), samotná zpráva a volitelně název modulu. Název modulu může představovat jakýkoli řetězec usnadňující identifikaci části aplikace, jež vygenerovala zprávu. Kód třídy `Logger` vypadá následovně:

```
<?php

class Logger {

  private $hLogFile;
  private $logLevel;

  //Úrovně protokolování. Čím vyšší číslo, tím menší vážnost zprávy
  //Mezery v číslování umožňují pozdější přidání dalších úrovní
  const DEBUG     = 100;
  const INFO      = 75;
  const NOTICE    = 50;
  const WARNING   = 25;
  const ERROR     = 10;
  const CRITICAL  = 5;

  //Soukromý konstruktor - třída používá návrhový vzor singleton
  private function __construct() {

    //Pseudokód, který načte konfiguraci
    //Implementace této metody, která by neměla být nijak obtížná, se ponechává
    //na čtenáři
    $cfg = Config::getConfig();

    /* Pokud konfigurace specifikuje výchozí úroveň, použije se. V opačném
       případě se jako výchozí úroveň nastaví úroveň INFO
    */
    $this->logLevel = isset($cfg['LOGGER_LEVEL'])?
                          $cfg['LOGGER_LEVEL']:
```

```
                        Logger::INFO;

  //Konfigurace musí specifikovat soubor protokolu
  if(! ( isset($cfg['LOGGER_FILE']) && strlen($cfg['LOGGER_FILE'])) ) {
    throw new Exception('Konfigurace nespecifikuje soubor protokolu.');
  }

  $logFilePath = $cfg['LOGGER_FILE'];

  //Otevření souboru protokolu. Chybové zprávy PHP se potlačí.
  //V případě chyby se vyhodí výjimka.
  $this->hLogFile = @fopen($logFilePath, 'a+');

  if(! is_resource($this->hLogFile)) {
    throw new Exception("Specifikovaný soubor protokolu $logFilePath " .
                'se nepodařilo otevřít pro zápis. Zkontrolujte nastavení ' .
                'přístupových práv.');
  }

  //Nastavení kódování na ISO-8859-2
  stream_encoding($this->hLogFile, 'iso-8859-2');
}

public function __destruct() {
  if(is_resource($this->hLogFile)) {
    fclose($this->hLogFile);
  }
}

public static function getInstance() {

  static $objLog;

  if(!isset($objLog)) {
    $objLog = new Logger();
  }

  return $objLog;
}

public function logMessage($msg, $logLevel = Logger::INFO, $module = null) {

  if($logLevel > $this->logLevel) {
    return;
  }
```

```
    /* Pokud jste v souboru php.ini pomocí parametru date.timezone
       nespecifikovali časové pásmo, nezapomeňte vložit do kódu řádek jako je
       ten následující. V opačném případě ho můžete vynechat.
    */
    date_default_timezone_set('America/New_York');

    $time = strftime('%x %X', time());
    $msg = str_replace("\t", '    ', $msg);
    $msg = str_replace("\n", ' ', $msg);

    $strLogLevel = $this->levelToString($logLevel);

    if(isset($module)) {
      $module = str_replace("\t", '    ', $module);
      $module = str_replace("\n", ' ', $module);
    }

    //formát protokolu: datum/čas úroveň zpráva názevModulu
    //jednotlivé položky jsou oddělené tabulátory, záznamy pak řádky
    $logLine = "$time\t$strLogLevel\t$msg\t$module\n";
    fwrite($this->hLogFile, $logLine);
  }

  public static function levelToString($logLevel) {
    switch ($logLevel) {
      case Logger::DEBUG:
        return 'Logger::DEBUG';
        break;
      case Logger::INFO:
        return 'Logger::INFO';
        break;
      case Logger::NOTICE:
        return 'Logger::NOTICE';
        break;
      case Logger::WARNING:
        return 'Logger::WARNING';
        break;
      case Logger::ERROR:
        return 'Logger::ERROR';
        break;
      case Logger::CRITICAL:
        return 'Logger::CRITICAL';
        break;
      default:
        return '[unknown]';
    }
```

```
    }
}
?>
```

Tato třída by se vám neměla zdát nijak komplikovaná. Všimněte si, že třída má soukromý konstruktor, aby se zabránilo vícenásobnému otevírání souboru v průběhu zpracování požadavku na stránku. Metoda `getInstance` umožňuje kódu užívajícímu tuto třídu získání její instance. Jedná se o návrhový vzor *singleton*, který jste mohli vidět už v kapitole 4.

Prvních pár řádků kódu definuje několik konstant specifikujících úrovně protokolování. Tyto konstanty se používají pro řízení toho, jaké informace se zaznamenávají. Při vývoji první verze aplikace, určené k dalšímu ladění, budete nejspíše chtít protokolovat celou řadu informací umožňujících snadnější dohledávání chyb. Poté, co se aplikace dočká produkční verze, množství protokolovaných informací se z důvodu úspory systémových prostředků zmenší. Protokolování menšího množství informací zvyšuje poměr signálu k šumu, tj. v daném počtu řádků protokolu se nachází více užitečných informací. Pokud aplikace běží v produkčním prostředí, většina ladicích informací získaných v průběhu vývoje nebude při dohledávání příčin problémů příliš k užitku, což ztěžuje dohledání těchto problémů v protokolu.

Konstruktor načítá z pole obsahujícího konfiguraci dvě základní informace: název souboru protokolu a aktuální úroveň protokolování. V tomto příkladu se předpokládá, že aplikace používá pouze jeden protokol. Později v této kapitole se dozvíte, jak rozšířit tuto třídu tak, aby poskytovala větší funkcionalitu a flexibilitu, co se záznamu informací týká.

Úroveň protokolování uložte do soukromé členské proměnné, která se později použije pro určení toho, jaké informace se mají zapisovat do protokolu. Dále se zkontroluje název souboru. Ověří se jeho zadání a také jestli uživatelský účet, pod kterým běží daný proces webového serveru, může skutečně otevřít, resp. Vytvořit, soubor protokolu. Parametr `a+` funkce `fopen` říká „otevři tento soubor pro připojení obsahu na konec souboru a pokud soubor neexistuje, vytvoř ho“. Handle souboru uložte jako další soukromou členskou proměnnou. Pokud se z jakéhokoli důvodu handle souboru nepodaří získat, vyhodí konstruktor výjimku.

Jakékoli volání metody `getInstance` by se mělo nacházet v bloku `try` z důvodu možného vzniku výjimky vyvolané nedostatečnými přístupovými právy k souboru protokolu nebo nedostatku volného místa na disku, kde se tento soubor nachází. Pokud k tomu dojde, neměla by celá aplikace přestat pracovat jen z důvodu protokolování. Výjimku byste měli zachytit a provést patřičnou akci, např. v podobě odeslání e-mailu administrátorům systému (robustní e-mailovou třídu, kterou k tomuto účelu můžete použít najdete v kapitole 11).

Tato třída obsahuje také destruktor zajišťující zavření souboru protokolu, je-li otevřený. Nebyl-li uveden platný soubor, nebude ani jeho handle platný. Přestože konstruktor v takovém případě vyhodí výjimku, dojde při uvolnění objektu z paměti k volání destruktoru. Z tohoto důvodu je třeba přidat dodatečný kód, který zajistí, aby nedošlo k vyvolání další chyby při pokusu o zavření neotevřeného souboru.

Metoda `getInstance` třídy `Logger` obsahuje statickou proměnnou zajišťující uložení instance třídy i po dokončení volání metody `getInstance`. Když se metoda zavolá poprvé, bude mít tato proměnná hodnotu `null`, což vyvolá vytvoření nové instance třídy `Logger`. Pokud uvedený soubor protokolu není platný, probublá výjimka až do této metody. Při následných voláních metody bude tato proměnná obsahovat referenci vytvořenou během prvního volání metody

a vrátí původní instanci třídy. Použití návrhového vzoru singleton zde zajišťuje úsporu systémových prostředků potřebných pro otevření a zavření souboru při vytváření, resp. destrukci instance třídy `Logger`.

Metoda `logMessage` odvede všechnu těžkou práci. Jako parametry bere text zprávy, úroveň zprávy a volitelně název modulu. Pokud je aktuální úroveň protokolování aplikace (uložená konstruktorem do soukromé členské proměnné `logLevel`) alespoň tak vysoká jako úroveň zprávy specifikovaná druhým parametrem metody, vytvoří se záznam v protokolu. Je-li aktuální úroveň protokolování aplikace menší než úroveň specifikovaná druhým parametrem, k vytvoření záznamu v protokolu nedojde. Tímto způsobem můžete řídit, jaké informace se do protokolu zapisují.

Společně se zprávou by se měla metodě `logMessage` předat také úroveň indikující vážnost zprávy. Pokud např. chcete pro ladicí účely vypsat do protokolu obsah určité proměnné, měl by mít druhý parametr metody `logMessage` hodnotu `Logger::DEBUG`. Při nasazení kódu do produkčního prostředí pak parametr `cfg['LOGGER_LEVEL']` nastavíte na úroveň `Logger::WARNING` nebo vyšší. Do protokolu se díky tomu přestanou zapisovat ladicí zprávy, sníží se zatížení procesoru a obsah protokolu se omezí na chyby určité vážnosti.

Metoda `logMessage` generuje informace o datu a čase pomocí funkce PHP s názvem `strftime`, která jako první parametr bere formátovací řetězec a jako druhý parametr unixové časové razítko. Formátovací řetězec použitý v této ukázce používá zkrácenou reprezentaci data (`%x`) a zkrácenou reprezentaci systémového času (`%X`). Např. na serveru nacházejícím se ve Spojených státech by funkce `strftime('%x %X', time())` vrátila 07/14/2008 03:55:25 (14. července 2008 v 3:55:25 ráno). Pokud se váš počítač nachází v jiné části světa, zobrazí se reprezentace data adekvátně vaší lokaci (např. 14.07.2008).

> Pokud chcete mít nad formátováním časového razítka větší kontrolu, prostudujte dokumentaci funkce `strftime`, nacházející se na adrese `www.php.net/strftime`. Pokud pro svůj protokol potřebujete jemnější časové rozlišení než je jedna sekunda, můžete použít funkci `microtime` s přesností v řádu milisekund. Více informací opět najdete v dokumentaci dané funkce.

Jazyk PHP 6 je při určování aktuální časové zóny o něco přísnější. Webový server se často nemusí nacházet ve stejné časové zóně jako programátor a mnoho serverů navíc nemá správně nastavenou časovou zónu odpovídající aktuální lokaci. Důsledkem je, že by každá aplikace měla specifikovat svou časovou zónu buďto přímo v souboru `php.ini`, anebo pomocí funkce `date_default_timezone_set`. Tato funkce se zpravidla volá jako součást konfigurace aplikace v úvodu každého požadavku na zobrazení stránky (namísto při volání konkrétní funkce, jak ukazuje předchozí příklad).

Poté, co se vytvoří textová reprezentace časového razítka, musí metoda `logMessage` naformátovat zprávu. Protože je vaším záměrem vytvoření strojově čitelného protokolu, oddělte od sebe položky na řádku pomocí tabulátoru (`\t`). Jednotlivé záznamy protokolu jsou od sebe oddělené znakem nového řádku (`\n`). Kvůli použitému formátování musíte ze zprávy odstranit jakékoli tabulátory a znaky konce řádků, které by se zde mohly vyskytovat. Způsobily by strukturální chybu protokolu, která by značně ztížila jeho parsování. Tabulátory nahraďte čtyřmi mezerami (výchozí počet znaků, které tabulátor ve většině textových editorů obsadí), konce řádků jednou mezerou. Pokud byl zadán parametr `module`, provedou se tyto náhrady i pro jeho hodnotu. Po-

kud protokolujete informace, pro které jsou tabulátory a konce řádků klíčové, a nelze je odstranit, zvažte jejich náhradu standardními zástupnými znaky pro `\t` a `\n`, jež umožní obnovení původní hodnoty při analýze dat.

Další zajímavou věcí, která se zde odehrává, je konverze konstanty pro úroveň protokolování na řetězec. Pokud někde v kódu PHP uvedete název konstanty, dosadí se její hodnota. Neexistuje tedy žádný zjevný způsob, jak převést konstantu na textovou reprezentaci názvu sebe samé. Z tohoto důvodu jsou textové reprezentace napevno uvedené v metodě `levelToString`. Blok `switch` projde všemi konstantami a vrátí odpovídající název konstanty. Ačkoli příkazy `break` nejsou technicky vzato nutné, jsou zde pro přehlednost. Stejně tak není vyloženě nutné deklarovat metodu jako statickou, ale protože nezávisí na žádné členské proměnné a můžete potřebovat zobrazit úroveň protokolování kódem vně třídy, díky deklaraci metody jako abstraktní můžete použít např. kód `echo Logger::levelToString($cfg['LOGGER_LEVEL'])`.

Následující kód ukazuje, jakým způsobem byste mohli použít třídu `Logger` v reálné aplikaci:

```
<?php
require_once('class.Config.php');
require_once('class.Logger.php');

//Implementace této třídy se opět ponechává na uživateli

Config::addConfig('LOGGER_FILE', '/var/log/myapplication.log');
Config::addConfig('LOGGER_LEVEL', Logger::INFO);

$log = Logger::getInstance();

  if(isset($_GET['fooid'])) {

    //Nezapíše se do protokolu - úroveň protokolování je příliš vysoká
    $log->logMessage('fooid existuje', Logger::DEBUG);

    //Zapíše se do protokolu, protože LOG_INFO je výchozí úrovní protokolování
    $log->logMessage('Hodnota fooid je '.  $_GET['fooid']);

  } else {

    //Také toto se zapíše do protokolu, včetně názvu modulu
    $log->logMessage('Nezadali jste fooid',
                  Logger::CRITICAL,
                 "Modul Foo");

    throw new Exception('fooid chybí!');
}
?>
```

Pokud je proměnná `_GET['fooid']` nastavená, provedou se dvě volání metody `logMessage`, ale do protokolu se kvůli nastavené úrovni protokolování zapíše pouze jedna zpráva. Obsah protokolu bude vypadat následovně:

```
03/17/04  03:58:42  Logger::INFO  Hodnota fooid je 25
```

Zde můžete vidět časové razítko, úroveň zprávy a zprávu samotnou.

Pokud není proměnná `_GET['fooid']` nastavená, najdete v protokolu následující záznam:

```
03/19/04  05:30:07  Logger::CRITICAL  Nezadali jste fooid  Modul Foo
```

V tomto případě se stránce, která zapisuje do protokolu, nepředal parametr `fooid` a došlo k zaprotokolování zprávy o úrovni `Logger:CRITICAL`.

## Rozšíření třídy Logger

Textový soubor představuje pro protokol velmi vhodné úložiště dat. Existují doslova tisíce utilit, které můžete použít pro parsing a analýzu takovýchto dat – od jednoduchých utilit pro UNIX (jako je `sed` a `grep`) až po profesionální komerční software pro analýzu protokolů s cenou v řádech desítek tisíc dolarů. Někdy však textový soubor není tím nejvhodnějším datovým úložištěm. Můžete chtít použít relační databázi nebo integrovat systémový protokol vaší platformy v rámci centralizace všech vašich aplikačních protokolů. Kromě toho třída `Logger` ve své stávající podobě dokáže zapisovat pouze do jednoho souboru, bez ohledu na to, kde v aplikaci se volá. Za určitých okolností budete chtít oddělené protokoly pro různé části aplikace nebo různé prováděné úlohy. V této části kapitoly rozšíříte třídu `Logger` tak, aby podporovala jakékoli datové úložiště a také více protokolů v jedné aplikaci.

V kapitole 6 jste se dozvěděli, jak se s využitím třídy `PDO` připojovat k rozdílným databázovým systémům pouhou změnou připojovacího řetězce. Řetězec `mysql:host=localhost;dbname=mydb` zajistí připojení k lokálnímu databázovému systému MySQL, konkrétně k jeho databázi s názvem `mydb`. Pro připojení k lokálnímu databázovému systému PostgreSQL a jeho databázi `yourdb` by měl řetězec podobu `pgsql:host=localhost;dbname=yourdb`. Poté, co se pomocí daného řetězce naváže spojení, je již veškeré další použití třídy `PDO` shodné pro všechny systémy RDBMS, ke kterým se připojíte. Velmi podobný koncept můžete použít také pro třídu `Logger` a umožnit tak její použití s jakýmkoli typem datového úložiště.

Cílem vylepšené třídy `Logger` je nabídnout podobnou syntaxi pro připojení k datovému úložišti protokolu. Budete moci používat různá datová úložiště, a tím pádem také ukládat různé protokoly pro různé části aplikace. Nová třída `Logger` by měla obsahovat určitý repozitář protokolů, ke kterým je připojena. Měli byste např. mít možnost použít protokoly s názvy „chyby“ a „dotazy“, které budou ukládat chybové zprávy, resp. dotazy SQL. Následující ukázka kódu demonstruje, čeho je možné s vylepšenou třídou dosáhnout:

```
<?php

Logger::register('errors', 'file:///var/log/error.log');
Logger::register('app',
                 'pgsql://postgres@db/errors?table=applog&timestamp=dtlog&' .
                 'msg=smesg&level=slevel&module=smod');
```

```
$objQLog = Logger::getInstance('app');

$sql = "SELECT * FROM foo";
$objQLog->logMessage("Načítám všechna data");

try {
  Database->getInstance()->select($sql);
} catch (DBQueryException $e) {
  $objErrLog = Logger::getInstance('errors');
  $objErrLog->logMessage($e->getMessage(), LOGGER_CRITICAL);
}
?>
```

Může vám připadat matoucí, že tento kód vidíte dříve než novou třídu, kterou kód používá, ale někdy je jednodušší navrhnout třídu na základě toho, jak se má používat. Ve skutečné aplikaci byste první dva řádky umístili do globálně připojovaného souboru. Tyto dva řádky zakládají dva různé protokoly:

- `app` – protokol `app` bude ukládat zprávy v databázi PostgreSQL, v tabulce s názvem `applog` se specifikovanými sloupci pro časové razítko, zprávu, úroveň zprávy a název modulu.
- `errors` – protokol chyb bude běžný textový soubor v umístění `/var/log/error.log`.

V hlavní části aplikace se bude většina zpráv ukládat do protokolu `app` pro další analýzu. Jakékoli chyby, které se vyskytnou, se uloží do protokolu `errors` reprezentovaného textovým souborem. Textový soubor se používá z toho důvodu, že jednou z chyb může být i chyba spojení s databází.

## *Parsing připojovacího řetězce*

Prvním krokem pro vytváření nové třídy `Logger` je přidání funkcionality pro parsing řetězce ve formátu `schéma://uživatel:heslo@hostitel:port/cesta?dotaz#fragment`. Jazyk PHP naštěstí nabízí velmi šikovnou funkci, která právě tohle zajistí. Funkce `parse_url` bere jako svůj parametr řetězec reprezentující adresu URL a vrací asociativní pole komponent zadané adresy URL (klíče pole mají hodnoty `scheme`, `user`, `password`, `host`, `port`, `path`, `query` a `fragment`). Pokud některá z uvedených komponent neexistuje, nepřidá se do pole prvek s daným klíčem (naproti přidání prvku s hodnotou `null`).

Podívejte se na následující ukázku:

```
$url = "ftp://anonymous@ftp.gnu.org:21/pub/gnu/gcc";
$arParts = parse_url($url);
var_dump($arParts);

//zobrazí:
Array (
  [scheme] = > ftp
  [user] = > anonymous
  [host] = > ftp.gnu.org
  [port] = > 21
  [path] = > /pub/gnu/gcc
)
```

Všimněte si, že pole neobsahuje prvky s klíči `password`, `query` a `fragment`, protože se nenachází v zadané adrese URL.

## Úprava třídy Logger tak, aby používala připojovací řetězec

Třída `Logger` použije prvek pole s klíčem `scheme` pro určení toho, která obsluha se má použít. Metoda `register` se použije pro navázání nového spojení a jako své parametry bere kanonický název a adresu URL. Protože se metoda `register` v ukázkovém kódu volá staticky, je třeba použít nějakou pokročilou metodu, která umožní, aby obě metody, `register` i `getInstance`, pracovaly se stejnými informacemi.

Následující kód ukazuje novou metodu `register` a přepracovanou metodu `getInstance` a konstruktor. Tento kód ve skutečnosti ukazuje celou třídu `Logger`. Ty části, které doznaly změn, jsou vyznačené. Uložte tuto modifikovanou verzi do souboru `class.Logger-enhanced.php`:

```
<?php

class Logger {

  private $hLogFile;
  private $logLevel;

  //Úrovně protokolování. Čím vyšší číslo, tím menší vážnost zprávy
  //Mezery v číslování umožňují pozdější přidání dalších úrovní
  const DEBUG     = 100;
  const INFO      = 75;
  const NOTICE    = 50;
  const WARNING   = 25;
  const ERROR     = 10;
  const CRITICAL  = 5;

  //Soukromý konstruktor - třída používá návrhový vzor singleton
  private function __construct() {

  }

  public static function register($logName, $connectionString) {

    $urlData = parse_url($connectionString);

    if(! isset($urlData['scheme'])) {
      throw new Exception("Neplatný připojovací řetězec $connectionString");
    }

    @include_once('Logger/class.'. $urlData['scheme']. 'LoggerBackend.php');

    $className = $urlData['scheme']. 'LoggerBackend';
```

```
    if(! class_exists($className)) {
      throw new Exception('Pro schéma '. $urlData['scheme']. ' není '.
        'dostupná žádná obsluha');
    }

    $objBack = new $className($urlData);

    Logger::manageBackends($logName, $objBack);
  }

  public static function getInstance($name) {
    return Logger::manageBackends($name);
  }

  private static function manageBackends($name, LoggerBackend $objBack = null) {

    static $backEnds;

    if(! isset($backEnds)) {
      $backEnds = array();
    }

    if($objBack == null) {
      //načítání
      if(isset($backEnds[$name])) {
        return $backEnds[$name];
      } else {
        throw new Exception("Specifikovaný protokol $name není " .
                  'zaregistrovaný.');
      }

    } else {
      //přidávání
      $backEnds[$name] = $objBack;
    }
  }

  public static function levelToString($logLevel) {
    switch ($logLevel) {
      case Logger::DEBUG:
        return 'Logger::DEBUG';
        break;
      case Logger::INFO:
        return 'Logger::INFO';
        break;
      case Logger::NOTICE:
```

```
        return 'Logger::NOTICE';
        break;
      case Logger::WARNING:
        return 'Logger::WARNING';
        break;
      case Logger::ERROR:
        return 'Logger::ERROR';
        break;
      case Logger::CRITICAL:
        return 'Logger::CRITICAL';
        break;
      default:
        return '[unknown]';
    }
  }
}
?>
```

V předchozím kódu třída `Logger` připojuje soubor definující abstraktní třídu s názvem `LoggerBackend`. Tato třída slouží jaké základní třída pro třídy, jež zajistí samotný zápis dat do daného protokolu. Konstruktor třídy `Logger` nyní zeje prázdnotou, ale stále je deklarovaný jako soukromý, aby se zabránilo vytváření instancí třídy mimo metodu `getInstance`.

Nová statická metoda `register` je zodpovědná za vytvoření instance třídy `LoggerBackend` na základě schématu určeného z připojovacího řetězce, který metoda bere jako svůj druhý parametr. Za tímto účelem metoda nejdříve volá funkci `parse_url` pro určení schématu. Pokud v zadaném připojovacím řetězci není schéma uvedeno, vyhodí metoda `register` výjimku. Pokud je schéma uvedeno, provede se pokus o připojení souboru s názvem `class.[schéma].LoggerBackend.php` nacházejícího se v adresáři `Logger`, pomocí funkce `include_once`, a vytvoří se instance v něm definované třídy (konstruktoru se jako parametr předá pole `urlData`). Pokud se takový soubor nepodaří najít, vyhodí kód výjimku. Namísto funkce `require_once` se zde používá funkce `include_once`. To proto, že funkce `require_once` vyhodí fatální výjimku ukončující běh aplikace, pokud se uvedený soubor nenajde.

V případě úspěšného vytvoření instance třídy `LoggerBackend` se tato předá společně s kanonickým názvem (specifikovaným prvním parametrem metody `register`) soukromé metodě `manageBackends`. Důvodem existence této metody je skutečnost, že jsou všechny veřejné metody třídy `Logger` nyní statické. Třída tak nemůže používat žádné členské proměnné pro uložení instancí tříd obsluh. Metoda `manageBackends` obsahuje statickou proměnnou nahrazující členskou proměnnou. Pokud má metoda dva parametry, uloží objekt `LoggerBackend` do pole `backEnds` s využitím parametru `name` jako klíče. Je-li uvedený pouze jeden parametr (parametr `name`), vrátí metoda `manageBackends` objekt `LoggerBackend` (pokud nějaký existuje) specifikovaný parametrem `name`.

Nová metoda `getInstance` už nadále nevrací objekt `Logger` a namísto toho vrací objekt `LoggerBackend`. Tento objekt získá voláním metody `manageBackends` pouze s jedním parametrem – kanonickým názvem požadovaného protokolu. Pokud není protokol s tímto názvem zaregistrován anebo se nezdařilo vytvoření jeho instance metodou `register`, vyhodí metoda `manageBackends` výjimku, která probublá do metody `getInstance`.

### Třída LoggerBackend

Metody `manageBackends`, `register` i `getInstances` pracují s objekty `LoggerBackend`. Tato třída poskytuje konkrétní konstruktor (existující konstruktor, který lze zavolat) a abstraktní metodu s názvem `logMessage`, již si pravděpodobně pamatujete ze třídy `Logger`. Druhá z metod se, co se počtu a typů přijímaných parametrů týká, nezměnila. Metoda `logMessage` třídy `LoggerBackend` je však nyní abstraktní – nemá žádné tělo, jedná se pouze o deklaraci metody, kterou musí implementovat třídy dědící od třídy `LoggerBackend`. Konstruktor třídy bere jako svůj parametr pole vrácené funkcí `parse_url` a uloží ho do chráněné členské proměnné. Další metody třída `LoggerBackend` nedefinuje.

Uložte následující kód do souboru `Logger/class.LoggerBackend.php`:

```
<?php
abstract class LoggerBackend {
  protected $urlData;

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

  abstract function logMessage($message, $logLevel = Logger::INFO, $module);
}
?>
```

Samotná třída je deklarovaná jako abstraktní, protože obsahuje abstraktní metodu a instance třídy `LoggerBackend` by nenašla žádné praktické uplatnění. Až podtřídy třídy `LoggerBackend` budou vykonávat nějakou práci.

### Podtřídy třídy LoggerBackend

Aby mohla nová a vylepšená třída `Logger` něco dělat, je třeba vytvořit alespoň jednu podtřídu třídy `LoggerBackend`. Protože už máte kód, který zajišťuje protokolování informací do souboru, bude pro začátek vytvoření podtřídy třídy `LoggerBackend` s touto funkcionalitou velmi jednoduché.

Vytvořte soubor s názvem `class.fileLoggerBackend.php` (dbejte na správnou velikost písmen) a vložte do něj následující kód:

```
<?php

require_once('class.LoggerBackend.php');

class fileLoggerBackend extends LoggerBackend {

  private $logLevel;
  private $hLogFile;

  public function __construct($urlData) {

    parent::__construct($urlData);
```

```
    $this->logLevel = Config::getConfig('LOGGER_LEVEL');

    $logFilePath = $this->urlData['path'];

    if(! strlen($logFilePath)) {
      throw new Exception('Soubor protokolu není uveden v připojovacím ' .
                  'řetězci.');
    }

    print "Zaznamenávám data do $logFilePath";

    //Otevření souboru protokolu. Chybové zprávy PHP se potlačí.
    //V případě chyby se vyhodí výjimka.
    $this->hLogFile = @fopen($logFilePath, 'a+');

    if(! is_resource($this->hLogFile)) {
      throw new Exception("Specifikovaný soubor protokolu $logFilePath " .
                   'se nepodařilo otevřít pro zápis. Zkontrolujte nastavení ' .
                   'přístupových práv.');
    }

    //Nastavení kódování na ISO-8859-2
    stream_encoding($this->hLogFile, 'iso-8859-2');

  }

  public function logMessage($msg, $logLevel = LOGGER_INFO, $module = null) {
    if($logLevel > $this->logLevel) {
      return;
    }

    /* Pokud jste v souboru php.ini pomocí parametru date.timezone
       nespecifikovali časové pásmo, nezapomeňte vložit do kódu řádek jako je
       ten následující. V opačném případě ho můžete vynechat.
    */
    date_default_timezone_set('America/New_York');
    $time = strftime('%x %X', time());
    $msg = str_replace("\t", '    ', $msg);
    $msg = str_replace("\n", ' ', $msg);

    $strLogLevel = Logger::levelToString($logLevel);

    if(isset($module)) {
      $module = str_replace("\t", '    ', $module);
      $module = str_replace("\n", ' ', $module);
```

```
    }

    //formát protokolu: datum/čas úroveň zpráva názevModulu
    //jednotlivé položky jsou oddělené tabulátory, záznamy pak řádky
    $logLine = "$time\t$strLogLevel\t$msg\t$module\n";
    fwrite($this->hLogFile, $logLine);

  }
}
?>
```

Většina kódu by vám měla být povědomá. Kód téměř kompletně pochází z původní třídy `Logger`. Třída `fileLoggerBackend` bude sloužit pro obsluhu protokolů zaregistrovaných s využitím schématu `file://`. Zapisuje protokol do souboru specifikovaného komponentou `path` připojovacího řetězce. Pro registraci protokolu tohoto typu by měl připojovací řetězec vypadat např. `file:///var/log/app.log`. Komponentami jsou zde `file://` (zakončené dvěma lomítky) a `/var/log/app.log` (uvozené jedním lomítkem). Za názvem schématu jsou tedy uvedeny celkem tři lomítka.

Chcete-li tuto obsluhu použít, stačí vytvořit kód podobný tomu následujícímu (blízký tomu, který jste viděli na začátku této části):

```
<?php
  require_once('class.Logger.php');
  Logger::register('app', 'file:///var/log/applog.log');
  $log = Logger::getInstance('app');
  $log->logMessage('Test protokolu!', LOGGER_CRITICAL, 'test');
?>
```

Tím se zaregistruje protokol s názvem `app`, který se zapisuje do souboru `/var/log/applog.log`, jehož obsluhu zajišťuje třída `fileLoggerBackend`.

### *Protokolování do databáze*

Stejný proces vytvoření podtřídy třídy LoggerBackend můžete použít pro vytvoření protokolovacího mechanizmu do jakéhokoli datového úložiště. Následující kód ukazuje podtřídu pro databázový systém PostgreSQL, ale stejný princip je možné použít i pro jakoukoli jinou databázovou platformu.

Tato třída má podstatně delší konstruktor. Parametry připojení a názvy sloupců tabulky databáze, do které se budou zapisovat informace, by se měly uvést v připojovacím řetězci, s doplněním vhodných výchozích hodnot volitelných položek. Kód tak musí nejdříve zkontrolovat nastavení určitých hodnot, než začne vytvářet připojovací řetězec a názvy sloupců. Tento kód není příliš zajímavý, ale přesto ho zde najdete, abyste viděli, jak provést parsing připojovacího řetězce za účelem vytvoření flexibilního protokolovacího mechanizmu využívajícího databázový systém PostgreSQL. Následující kód uložte do souboru `Logger/class.pgsqlLoggerBackend.php`:

```
<?php


require_once('class.LoggerBackend.php');
```

```
class pgsqlLoggerBackend extends LoggerBackend {

  private $logLevel;
  private $hConn;
  private $table = 'logdata';
  private $messageField = 'message';
  private $logLevelField = 'loglevel';
  private $timestampField = 'logdate';
  private $moduleField = 'module';

  public function __construct($urlData) {

    parent::__construct($urlData);

    $this->logLevel = Config::getConfig('LOGGER_LEVEL');

    $host = $urlData['host'];
    $port = $urlData['port'];
    $user = $urlData['user'];
    $password = $urlData['pass'];
    $arPath = explode('/', $urlData['path']);
    $database = $arPath[1];

    if(!strlen($database)) {
      throw new Exception('pgsqlLoggerBackend: Neplatný připojovací řetězec.' .
                          'Název databáze není uveden.');
    }

    $connStr = '';
    if($host) {
      $connStr .= "host=$host ";
    }

    if($port) {
      $connStr .= "port=$port ";
    }
    if($user) {
      $connStr .= "user=$user ";
    }

    if($password) {
      $connStr .= "password=$password ";
    }

    $connStr .= "dbname=$database";
```

```
//Běžná chybová hlášení se potlačí. V případě chyby se vyhodí výjimka.
$this->hConn = pg_connect($connStr);

if(! is_resource($this->hConn)) {
  throw new Exception("Nepodařilo se připojení k databázi pomocí $connStr");
}

//Načtení řetězce ve formátu var=foo&bar=blah
//a jeho převod na pole
// array('var' => 'foo', 'bar' => 'blah')
//Hodnoty zakódované pro URL je třeba dekódovat
$queryData = $urlData['query'];
if(strlen($queryData)) {
  $arTmpQuery = explode('&',$queryData);

  $arQuery = array();
  foreach($arTmpQuery as $queryItem) {
    $arQueryItem = explode('=', $queryItem);
    $arQuery[urldecode($arQueryItem[0])] = urldecode($arQueryItem[1]);
  }
}

/* Ani jedna z následujících položek není povinná. Výchozí hodnoty se
   nastavují v soukromých členských proměnných na začátku třídy.
   Tyto hodnoty specifikují název tabulky a názvy sloupců této
   tabulky, do kterých se budou ukládat nejrůznější položky protokolu.
*/

if(isset($arQuery['table'])) {
  $this->table = $arQuery['table'];
}

if(isset($arQuery['messageField'])) {
  $this->messageField = $arQuery['messageField'];
}
   if(isset($arQuery['logLevelField'])) {
  $this->logLevelField = $arQuery['logLevelField'];
}

    if(isset($arQuery['timestampField'])) {
  $this->timestampField = $arQuery['timestampField'];
}
if(isset($arQuery['moduleField'])) {
  $this->logLevelField = $arQuery['moduleField'];
}
```