16

RPC: Interakce se vzdálenými službami

Jednoduše řečeno, služby vzdáleného volání procedur (RPC) poskytují standardizované rozhraní pro tvorbu volání funkcí nebo metod přes síť.

Prakticky každý aspekt webového programování obsahuje služby RPC. Na bázi RPC jsou požadavky HTTP provedené webovými prohlížeči na webové servery i dotazy zaslané databázovému serveru klienty databáze. I když oba tyto příklady jsou vzdálenými voláními, ve skutečnosti se nejedná o protokoly RPC. Postrádají generalizaci a standardizaci volání RPC; protokoly používané webovým serverem a databázovým serverem nelze například sdílet, přestože jsou vytvořeny na stejném síťovém protokolu.

Aby byl protokol RPC užitečný, měl by vykazovat následující vlastnosti:

- **Generalizace** – mělo by být snadné vkládat nové volatelné metody.
- **Standardizace** – jestliže znáte název a seznam parametrů metody, měli byste být schopni pro ni jednoduše sestavit požadavek.
- **Jednoduše konvertovatelný** – vrácená hodnota RPC by měla jít jednoduše zkonvertovat na příslušný nativní typ dat.

Samotný protokol HTTP nesplňuje žádné z těchto kritérií, ale poskytuje extrémně vhodnou transportní vrstvu pro zasílání požadavků RPC. V této kapitole se zaměříme na dva nejoblíbenější protokoly RPC a to XML-RPC a SOAP, které se tradičně implementují přes web.

Použití protokolů RPC v aplikacích s velkým provozem

I když jsou RPC extrémně flexibilními nástroji, vnitřně jsou pomalé. Jakýkoli proces, který využívá RPC se ihned prováže s výkonem a dostupností vzdálené služby. Dokonce i v nejlepším případě zaznamenáte zdvojnásobení času služby na každé obsluhované stránce. Jsou-li na vzdáleném koncovém bodu nějaké poruchy, celý web se může viset na dotazech RPC. U administrativních nebo málo využívaných služeb to nepředstavuje velký problém, avšak u produkčních nebo velmi navštěvovaných stránek to většinou není přijatelné.

Kouzelné řešení minimalizace dopadu na produkční služby a redukce problémů latence a dostupnosti webových služeb spočívá v implementaci cachovací strategie, díky níž se zbavíme přímé závislosti na vzdálené službě. O cachovacích strategií, které lze jednoduše adaptovat pro zpracování volání RPC, jsme hovořili v kapitole 10 a 11.

XML-RPC

XML-RPC je předchůdcem protokolů RPC založených na XML. XML-RPC je nejčastěji zapouzdřen v požadavku a odezvě `POST` protokolu HTTP, ačkoli, jak si v krátkosti řekneme v kapitole 15, není to nezbytně nutné. Jednoduchý požadavek XML-RPC představuje dokument XML, který vypadá asi takto:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
<methodName>system.load</methodName>
<params>
</params>
</methodCall>
```

Tento požadavek se odešle metodou `POST` na server XML-RPC. Server následně vyhledá a provede specifikovanou metodu (v tomto případě `system.load`) a předá specifikované parametry (v tomto případě se nepředají žádné parametry). Výsledek se poté předá zpět volajícímu. Vrácená hodnota tohoto požadavku představuje řetězec, který obsahuje aktuální zatížení počítače, získané z výsledku unixového příkazu `uptime`. Podívejte se na příklad výstupu:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
<params>
<param>
<value>
<string>0.34</string>
</value>
</param>
</params>
</methodResponse>
```

Tyto dokumenty samozřejmě nemusíte sami vytvářet a interpretovat. Pro PHP existuje celá řada různých implementací XML-RPC. Obecně preferuji použití tříd PEAR XML-RPC, protože jsou distribuovány se samotným PHP. (Používá je instalátor PEAR.) Tudíž mají téměř 100% implementaci. Díky tomu existují pouze pramalé důvody poohlížet se po něčem jiném. Dialog XML-RPC se skládá ze dvou částí: požadavek klienta a odezva serveru.

Nejprve si něco řekneme o kódu klienta. Klient vytvoří dokument `request`, zašle jej na server a analyzuje (parse) odezvu. Následující kód generuje dokument požadavku znázorněný dříve v této sekci a analyzuje výslednou odezvu:

```
require_once 'XML/RPC.php';
$client = new XML_RPC_Client('/xmlrpc.php', 'www.example.com');
$msg = new XML_RPC_Message('system.load');
$result = $client->send($msg);
if ($result->faultCode()) {
print "Error\n";
}
print XML_RPC_decode($result->value());
```

Vytvořte nový objekt `XML_RPC_Client`, který předává vzdálené službě URI a adresu.

Poté se vytvoří `XML_RPC_Message` obsahující název metody, která se má vyvolat (v tomto případě `system.load`). Protože této metodě nejsou předány žádné parametry, není potřeba do zprávy vkládat další data.

Dále se přes metodu `send()` odešle zpráva na server. Výsledek se zkontroluje, zdali neobsahuje chybu. Není-li výsledkem chyba, hodnota výsledku se dekóduje z formátu XML do nativního typu PHP a vytiskne se pomocí `XML_RPC_decode()`.

Pro příjem požadavku, nalezení a provedení příslušného zpětného volání a vrácení odezvy musíte podporovat funkcionalitu na straně serveru. Podívejte se na příklad implementace, která zpracuje metodu `system.load`, kterou jste požadovali v kódu klienta:

```
require_once 'XML/RPC/Server.php';
function system_load()
{
$uptime = `uptime`;
if(preg_match("/load average: ([\d.]+)/", $uptime, $matches)) {
return new XML_RPC_Response( new XML_RPC_Value($matches[1], 'string'));
}
}
$dispatches = array('system.load' => array('function' => 'system_uptime'));
new XML_RPC_Server($dispatches, 1);
```

Jelikož funkce PHP potřebné pro podporu příchozích požadavků jsou definovány, stačí pouze zpracovat `system.load request`, který se implementuje přes funkci `system_load()`. Tato funkce spustí

příkaz `uptime` a extrahuje z počítače jednominutovou průměrnou zátěž. Dále serializuje extrahovanou zátěž do `XML_RPC_Value` a zabalí ji do `XML_RPC_Response` pro vrácení uživateli.

Dále se registruje funkce zpětného volání v odesílací mapě, která informuje server o tom, jak odeslat příchozí požadavky do specifických funkcí. Vytvoříte pole funkcí `$dispatches`, které se vyvolá. Tohle je pole, které mapuje názvy metod XML-RPC na názvy funkcí PHP. Nakonec se vytvoří objekt `XML_RPC_Server` a předá se mu odesílací pole `$dispatches`. Druhý parametr `1` signalizuje, že by se měl požadavek obsloužit okamžitě s pomocí metody `service()` (která se volá interně).

Metoda `service()` hledá nezpracovaná data `POST HTTP`, analyzuje je pro požadavek XML-RPC a poté provede odeslání. Protože se spoléhá na autoglobální proměnnou PHP `$HTTP_RAW_POST_DATA`, musíte se ujistit, že máte v souboru `php.ini` zapnuto `always_populate_raw_post_data`.

Nyní byste měli obdržet následující, pokud umístíte kód serveru na `www.example.com/xmlrpc.php` a z libovolného počítače provedete kód klienta:

```
> php system_load.php
0.34
```

případně jinou hodnotu udávající vaše průměrné jednominutové zatížení.

Tvorba serveru: implementace MetaWeblog API

Síla XML-RPC spočívá v tom, že poskytuje standardizovanou metodu pro komunikaci mezi službami. To je užitečné zejména, když neovládáte oba konce požadavku služby. XML-RPC vám umožní jednoduše vytvořit dobře definované rozhraní na poskytovanou službu. Příkladem toho je odesílací API web ogu.

K dispozici je celá řada systémů pro blogování na webu a existuje spousta nástrojů, které pomáhají lidem uspořádat a zaslat záznamy těmto systémům. Pokud by neexistovaly standardizované procedury, každý nástroj by musel podporovat každý weblog, aby byl obecně použitelný nebo každý weblog by musel podporovat všechny nástroje. Tento druh spleti relací by nebylo možné škálovat.

I když se množina funkcí a implementace weblogových systémů značně liší, je možné definovat množinu standardních operací, které je potřeba provést k odeslání záznamů systémům pro blogování na webu. A následně, weblogy a nástroje musí implementovat pouze toto rozhraní, aby byly nástroje kompatibilní se všemi systémy pro blogování na webu.

Na rozdíl od velkého počtu systémů pro blogování, které jsou k dispozici, se obecně se používají pouze tři odesílací API weblogů: Blogger API, MetaWeblog API a MovableType API (což ve skutečnosti je pouze rozšíření MetaWeblog API). Všechny dostupné nástroje pro vytváření weblogů používají jeden z těchto tří protokolů, a proto, pokud implementujete tyto API, váš weblog bude schopen komunikovat s jakýmkoliv nástrojem. Tohle je obrovská výhoda pro vytváření nových snadno adaptovatelných blogovacích systémů.

Samozřejmě nejprve musíte mít webový blogovací systém, který lze zacílit jedním z API. Vytváření celého systému přesahuje rámec této kapitoly, a proto, místo jeho vytváření od úplného začátku, můžeme

vložit vrstvu XML-RPC do blogovacího systému Serendipity. Zmíněné API zpracuje odeslání, a proto bude možné sestavit rozhraní s následujícími rutinami z Serendipity:

```
function serendipity_updertEntry($entry) {}
function serendipity_fetchEntry($key, $match) {}
```

`serendipity_updertEntry()` je funkce, která buď aktualizuje stávající záznam nebo vloží nový záznam v závislosti na tom, zda je předáno `id`. Její parametr `$entry` je pole, které přestavuje řádkovou bránu (row gateway) (elementy pole jsou se sloupci tabulky v poměru jeden-k-jednomu) pro následující tabulku databáze:

```
CREATE TABLE serendipity_entries (
id INT AUTO_INCREMENT PRIMARY KEY,
title VARCHAR(200) DEFAULT NULL,
timestamp INT(10) DEFAULT NULL,
body TEXT,
author VARCHAR(20) DEFAULT NULL,
isdraft INT
);
```

`serendipity_fetchEntry()` načte záznam z tabulky porovnáním specifikované dvojice klíč/hodnota.

Cílem naší implementace je MetaWeblog API, protože poskytuje větší hloubku funkcionalit než Blogger API. MetaWeblog API implementuje tři hlavní metody:

```
metaWeblog.newPost(blogid,username,password,item_struct,publish) returns string
metaWeblog.editPost(postid,username,password,item_struct,publish) returns true
metaWeblog.getPost(postid,username,password) returns item_struct
```

`blogid` je identifikátor weblogu, na který se zaměřujete (což je užitečné, pokud systém podporuje více samostatných weblogů). `username` a `password` jsou autentizační kritéria, které identifikují odesílatele. `publish` je příznak, zdali záznam je koncept nebo by se měl publikovat živě.

`item_struct` je pole dat k odeslání.

Místo implementace nového formátu dat pro záznamy se autor specifikace MetaWeblogu – Dave Winer – rozhodl použít definici položky `item` ze specifikace RSS (Really Simple Syndication) 2.0, k dispozici na adrese `http://blogs.law.harvard.edu/tech/rss`. RSS je standardizovaný formát XML vytvořený pro prezentování článků a záznamů v novinách. Jeho uzel `item` obsahuje následující elementy:

Element	Popis
`title`	Titulek `item`.
`link`	URL, které odkazuje na naformátovaný tvar položky.
`description`	Shrnutí položky.
`author`	Název autora položky. Podle specifikace RSS by to měla být emailová adresa, ačkoli mnohem častěji se používá přezdívka.
`pubDate`	Datum publikace záznamu.

Specifikace rovněž volitelně poskytuje pole pro odkazy na vlákna komentáře, jedinečné identifikátory a kategorie. Navíc spousta weblogů rozšiřuje RSS definici `item` pro zahrnutí elementu `content:encoded` obsahující úplnou poštu, nejenom tedy shrnutí pošty, které se tradičně nachází v RSS elementu `description`.

Chcete-li zavést MetaWeblog API, musíte definovat funkce pro implementace tří zmíněných metod. Nejprve se podívejme na funkci pro zpracování zasílání nových záznamů:

```
function metaWeblog_newPost($message) {
$username = $message->params[1]->getval();
$password = $message->params[2]->getval();
if(!serendipity_authenticate_author($username, $password)) {
return new XML_RPC_Response('', 4, 'Authentication Failed');
}
$item_struct = $message->params[3]->getval();
$publish = $message->params[4]->getval();
$entry['title'] = $item_struct['title'];
$entry['body'] = $item_struct['description'];
$entry['author'] = $username;
$entry['isdraft'] = ($publish == 0)?'true':'false';
$id = serendipity_updertEntry($entry);
return new XML_RPC_Response( new XML_RPC_Value($id, 'string'));
}
```

`metaWeblog_newPost()` extrahuje z požadavku parametry `username` a `password` a s pomocí metody `getval()` deserializuje jejich XML reprezentace do typů PHP. Poté `metaWeblog_newPost()` autentizuje specifikovaného uživatele. Pokud uživatel neprokáže svou identitu, `metaWeblog_newPost()` vrací prázdný objekt `XML_RPC_Response` s chybovou zprávou "`Authentication Failed`".

Proběhne-li autentizace úspěšně, `metaWeblog_newPost()` načte parametr `item_struct` a s pomocí metody `getval()` jej deserializuje na PHP pole `$item_struct`. Pole `$entry` definující reprezentaci interního záznamu Serendipity se sestaví z `$item_struct` a předá se `serendipity_updertEntry()`. `XML_RPC_Response`, kterou tvoří identifikátor nového záznamu, se vrátí volajícímu.

Back-end pro `MetaWeblog.editPost` se velmi podobá `MetaWeblog.newPost`. Podívejte se na kód:

```
function metaWeblog_editPost($message) {
$postid = $message->params[0]->getval();
$username = $message->params[1]->getval();
$password = $message->params[2]->getval();
if(!serendipity_authenticate_author($username, $password)) {
return new XML_RPC_Response('', 4, 'Authentication Failed');
}
$item_struct = $message->params[3]->getval();
$publish = $message->params[4]->getval();
$entry['title'] = $item_struct['title'];
$entry['body'] = $item_struct['description'];
$entry['author'] = $username;
$entry['id'] = $postid;
$entry['isdraft'] = ($publish == 0)?'true':'false';
$id = serendipity_updertEntry($entry);
return new XML_RPC_Response( new XML_RPC_Value($id?true:false, 'boolean'));
}
```

Provede se stejná autentizace a provede se sestavení a aktualizace pole `$entry`. Pokud `serendipity_updertEntry` vrací `$id`, znamená to, že byla úspěšná a odezva se nastaví na hodnotu `true`, jinak se odezva nastaví na hodnotu `false`.

Poslední funkcí, kterou máme ještě implementovat, je zpětné volání `MetaWeblog.getPost`. Tato funkce používá `serendipity_fetchEntry()` k získání podrobných informací o poště a poté naformátuje odezvu XML obsahující `item_struct`. Podívejte se na implementaci:

```
function metaWeblog_getPost($message) {
$postid = $message->params[0]->getval();
$username = $message->params[1]->getval();
$password = $message->params[2]->getval();
if(!serendipity_authenticate_author($username, $password)) {
return new XML_RPC_Response('', 4, 'Authentication Failed');
}
$entry = serendipity_fetchEntry('id', $postid);
$tmp = array(
'pubDate' => new XML_RPC_Value(
XML_RPC_iso8601_encode($entry['timestamp']), 'dateTime.iso8601'),
'postid' => new XML_RPC_Value($postid, 'string'),
'author' => new XML_RPC_Value($entry['author'], 'string'),
'description' => new XML_RPC_Value($entry['body'], 'string'),
'title' => new XML_RPC_Value($entry['title'],'string'),
'link' => new XML_RPC_Value(serendipity_url($postid), 'string')
);
$entry = new XML_RPC_Value($tmp, 'struct');
```

```
return new XML_RPC_Response($entry);
}
```

Všimněte si, že po načtení záznamu se v item připraví pole všech dat. `XML_RPC_iso8601()` se postará o formátování časové známky Unixu, kterou používá Serendipity, na formát odpovídající ISO 8601, jenž vyžaduje RSS `item`. Výsledné pole se poté serializuje jako `struct XML_RPC_Value`. Tohle představuje standardní způsob sestavování XML-RPC typu struct ze základních typů PHP.

Doposud jste viděli, že `XML_RPC_Value` lze předat identifikátory `string`, `boolean`, `dateTime`.iso8601 a `struct`. Podívejte se na úplný seznam všech možností:

Typ	Popis
`i4/int`	32bitové celé číslo
`boolean`	logický typ
`double`	číslo s plovoucí desetinou čárkou
`string`	řetězec
`dateTime.iso8601`	časová známka ve formátu ISO 8601
`base64`	základní 64kódovaný řetězec
`struct`	implementace asociativního pole
`array`	neasociativní (indexované) pole

Typy `struct` a `array` mohou obsahovat jakýkoli datový typ (včetně dalších elementů `struct` a `array`). Není-li specifikován žádný typ, použije se `string`. I když všechna PHP data lze reprezentovat buď jako `string`, `struct` nebo `array`, jsou podporovány i ostatní typy, protože vzdálené aplikace napsané v jiných jazycích mohou vyžadovat specifičtější typ dat.

Chcete-li registrovat tyto funkce, vytvořte odeslání následovně:

```
$dispatches = array(
metaWeblog.newPost' =>
array('function' => 'metaWeblog_newPost'),
'metaWeblog.editPost' =>
array('function' => 'metaWeblog_editPost'),
'metaWeblog.getPost' =>
array('function' => 'metaWeblog_getPost'));
$server = new XML_RPC_Server($dispatches,1);
```

Gratulujeme! Váš software je nyní kompatibilní s MetaWeblog API!

Informativní služby XML-RPC

Uživatelé služeb XML-RPC jistě ocení možnost požádat server o podrobné informace o všech poskytovaných službách. XML-RPC definují tři standardní, vestavěné metody pro takovéto sebepozorování:

- **system.listMethods** – vrací pole všech metod implementovaných serverem (všechny zpětné volání registrované v odesílací mapě).
- **system.methodSignature** – přijímá jeden parametr (název metody) a vrací pole možných signatur (prototypů) metody.
- **system.methodHelp** – přijímá název metody a vrací dokumentační řetězec pro metodu.

Protože PHP je dynamický jazyk a nedbá na dodržování počtu nebo typu argumentů předávaných funkci, data, která se mají vrátit pomocí `system.methodSignature`, musí být specifikovány uživatelem. Metody v XML-RPC mohou mít různé parametry, proto vrácená množina je pole všech možných signatur. Každá signatura je sama o sobě polem; první element pole je návratový typ metody a zbývající elementy představují parametry metody.

Pro poskytování těchto dalších informací musí server zvětšit svou odesílací mapu, aby zahrnul další informace, jak můžete vidět níže u metody `metaWeblog.newPost`:

```
$dispatches = array(
'metaWeblog.newPost' =>
array('function' => 'metaWeblog_newPost',
'signature' => array(
array($GLOBALS['XML_RPC_String'],
$GLOBALS['XML_RPC_String'],
$GLOBALS['XML_RPC_String'],
$GLOBALS['XML_RPC_String'],
$GLOBALS['XML_RPC_Struct'],
$GLOBALS['XML_RPC_String']
)
),
'docstring' => 'Takes blogid, username, password, item_struct '.
'publish_flag and returns the postid of the new entry'),
/* ... */
);
```

Tyto tři metody můžete spolu kombinovat pro získání uceleného obrázku o tom, co implementuje server XML-RPC. Podívejte se na skript, který vypíše dokumentaci a signatury pro každou metodu na daném serveru XML-RPC:

```
<?php
require_once 'XML/RPC.php';
if($argc != 2) {
print "Must specify a url.\n";
exit;
}
$url = parse_url($argv[1]);
$client = new XML_RPC_Client($url['path'], $url['host']);
```

```
$msg = new XML_RPC_Message('system.listMethods');
$result = $client->send($msg);
if ($result->faultCode()) {
echo "Error\n";
}
$methods = XML_RPC_decode($result->value());
foreach($methods as $method) {
$message = new XML_RPC_Message('system.methodSignature',
array(new XML_RPC_Value($method)));
$response = $client->send($message)->value();
print "Method $method:\n";
$docstring = XML_RPC_decode(
$client->send(
new XML_RPC_Message('system.methodHelp',
array(new XML_RPC_Value($method))
)
)->value()
);
if($docstring) {
print "$docstring\n";
}
else {
print "NO DOCSTRING\n";
}
$response = $client->send($message)->value();
if($response->kindOf() == 'array') {
$signatures = XML_RPC_decode($response);
for($i = 0; $i < count($signatures); $i++) {
$return = array_shift($signatures[$i]);
$params = implode(", ", $signatures[$i]);
print "Signature #$i: $return $method($params)\n";
}
} else {
print "NO SIGNATURE\n";
}
print "\n";
}
?>
```

Spustíte-li skript na instalaci Serendipity, vygeneruje se následující výstup:

```
> xmlrpc-listmethods.php http://www.example.org/serendipity_xmlrpc.php
/* ... */
```

```
Method metaWeblog.newPost:
Takes blogid, username, password, item_struct, publish_flag
and returns the postid of the new entry
Signature #0: string metaWeblog.newPost(string, string, string, struct, string)
/* ... */
Method system.listMethods:
This method lists all the methods that the XML-RPC server knows
how to dispatch
Signature #0: array system.listMethods(string)
Signature #1: array system.listMethods()
Method system.methodHelp:
Returns help text if defined for the method passed, otherwise
returns an empty string
Signature #0: string system.methodHelp(string)
Method system.methodSignature:
Returns an array of known signatures (an array of arrays) for
the method name passed. If no signatures are known, returns a
none-array (test for type != array to detect missing signature)
Signature #0: array system.methodSignature(string)
```

SOAP

SOAP byla původně zkratka Simple Object Access Protocol, ale od verze 1.1 se jedná už pouze o označení, a nikoliv o zkratku. SOAP je protokol pro výměnu dat v heterogenním prostředí. Na rozdíl od XML-RPC, který je určen především pro zpracování RPC, je SOAP vytvořen pro generické zasílání zpráv a RPC je pouze jednou z aplikací SOAP. Je však potřeba říct, že tato kapitola pojednává o RPC, přičemž se soustředíme pouze na podmnožinu SOAP 1.1 používanou k implementaci RPC.

Jak tedy SOAP vypadá? Podívejte se na vzorek obalu SOAP, který používá `xmeth-ods.net` – vzorovou službu SOAP pro implementaci kanonického příkladu SOAP RPC pro získávání skladových cen IBM (kanonický příklad proto, že jde o příklad z návrhového dokumentu SOAP):

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<soap:Body>
<getQuote xmlns="http://www.themindelectric.com/wsdl/net.xmethods.services.
stockquote.StockQuote/">
<symbol xsi:type="xsd:string">ibm</symbol>
```

```
</getQuote>
</soap:Body>
</soap:Envelope>
Tohle je odezva:
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<soap:Body>
<n:getQuoteResponse xmlns:n="urn:xmethods-delayed-quotes">
<Result xsi:type="xsd:float">90.25</Result>
</n:getQuoteResponse>
</soap:Body>
</soap:Envelope>
```

SOAP je vynikající příklad skutečnosti, že jednoduchý koncept nemusí vždy přinášet jednoduchou implementaci. Zprávu SOAP tvoří obal, který obsahuje hlavičku a tělo. V SOAP má všechno jmenný prostor, což je teoreticky dobrá věc, i když čtení XML je poté obtížnější.

Nejvyšší uzel je `Envelope`, což je kontejner zprávy SOAP. Tohle je element ve jmenném prostoru `xmlsoap`, jak udává jeho plně kvalifikovaný název elementu `<soap:Envelope>` a deklarace jmenného prostoru:

```
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
```

která tvoří asociaci mezi `soap` a jmenným prostorem URI `http://schemas.xmlsoap.org/soap/envelope/`.

SOAP a Schema

SOAP znesnadňuje použití Schema, což je jazyk založený na XML pro definici a validaci datových struktur. Úplný jmenný prostor elementu je obvykle (například `http://schemas.xmlsoap.org/soap/envelope/`) dokument Schema, který popisuje jmenný prostor. Není to však podmínkou – jmenný prostor dokonce nemusí být URL.

Jmenné prostory plní stejný účel v XML jako v jakémkoli programovacím jazyku: zabraňují možným kolizím dvou názvů. Popřemýšlejte o nejvyšším uzlu `<soap-env:Envelope>`. Název atributu `Envelope` je ve jmenném prostoru `soap-env`. Takže, pokud by z nějakého důvodu chtěl FedEX definovat formát XML, který používá `Envelope` jako atribut, mohlo by to být `<FedEX:Envelope>` a všichni by byli spokojeni.

Zde jsou čtyři jmenné prostory deklarované v SOAP `Envelope`:

- **xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"** – obal SOAP definice Schema popisuje základní objekty SOAP a je standardním jmenným prostorem obsaženým v každém požadavku SOAP.
- **xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"** – atribut elementu `xsi:type` se převážně používá pro specifikaci typů elementů.
- **xmlns:xsd="http://www.w3.org/2001/XMLSchema"** – Schema deklaruje počet základních datových typů, které lze použít pro specifikaci a validaci.
- **xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"** – specifikace typu kódování použitého ve standardních požadavcích SOAP.

Element `<GetQuote>` má rovněž jmenný prostor – v tomto případě s následujícím extrémně dlouhým názvem:

```
http://www.themindelectric.com/wsdl/net.xmethods.services.stockquote.Stock-
Quote
```

Všimněte si, jak je použito Schema pro specifikaci typu a uspořádání skladového symbolu, který se předává:

```
<symbol xsi:type="xsd:string">ibm</symbol>
```

A podobně – v odezvě uvidíte specifický zápis skladové ceny:

```
<Result xsi:type="xsd:float">90.25</Result>
```

Výše uvedené specifikuje, že výsledek musí být číslo s plovoucí desetinou tečkou. To je užitečné, protože jsou zde sady nástrojů pro validaci Schema, která vám umožní ověřit váš dokument. Mohly by vám sdělit, že odezva v tomto formuláři není platná, protože `foo` není platnou reprezentací čísla s plovoucí desetinou tečkou:

```
<Result xsi:type="xsd:float">foo</Result>
```

WSDL

SOAP je doplněno jazykem WSDL (Web Services Description Language). WSDL je jazyk založený na XML popisující schopností a metody interakce s webovými službami (častěji než SOAP). Zde je soubor WSDL, který popisuje službu, na kterou jsou v předchozí sekci sestaveny požadavky:

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="net.xmethods.services.stockquote.StockQuote"
targetNamespace="http://www.themindelectric.com/wsdl/net.xmethods.services.
stockquote.StockQuote/"xmlns:tns="http://www.themindelectric.com/wsdl/net.
xmethods.services.stockquote.StockQuote/"
xmlns:electric="http://www.themindelectric.com/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```
xmlns="http://schemas.xmlsoap.org/wsdl/">
<message name="getQuoteResponse1">
<part name="Result" type="xsd:float" />
</message>
<message name="getQuoteRequest1">
<part name="symbol" type="xsd:string" />
</message>
<portType name="net.xmethods.services.stockquote.StockQuotePortType">
<operation name="getQuote" parameterOrder="symbol">
<input message="tns:getQuoteRequest1" />
<output message="tns:getQuoteResponse1" />
</operation>
</portType>
<binding name="net.xmethods.services.stockquote.StockQuoteBinding"
type="tns:net.xmethods.services.stockquote.StockQuotePortType">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
<operation name="getQuote">
<soap:operation soapAction="urn:xmethods-delayed-quotes#getQuote" />
<input>
<soap:body use="encoded" namespace="urn:xmethods-delayed-quotes"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</input>
<output>
<soap:body use="encoded" namespace="urn:xmethods-delayed-quotes"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</output>
</operation>
</binding>
<service name="net.xmethods.services.stockquote.StockQuoteService">
<documentation>
net.xmethods.services.stockquote.StockQuote web service
</documentation>
<port name="net.xmethods.services.stockquote.StockQuotePort"
binding="tns:net.xmethods.services.stockquote.StockQuoteBinding">
<soap:address location="http://66.28.98.121:9090/soap" />
</port>
</service>
</definitions>
```

WDSL se zřetelně zapojuje do obtížného použití jmenných prostorů, přičemž uspořádání má poněkud zmatenou logiku.

První částí tohoto kódu, která stojí za povšimnutí, je uzel `<portType>`. Tento uzel specifikuje činnosti, které lze provést, a vstupní a výstupní zprávy. Definuje to `getQuote`, která přijímá `getQuoteRequest1` a odpovídá pomocí `getQuteResponse1`.

Uzly `<message>` pro `getQuoteResponse1` specifikují, že obsahuje jediný element `Result` typu `float`. A podobně, `getQuoteRequest1` musí obsahovat jediný element symbol typu `string`.

Dále je tu uzel `<binding>`. Vazba se asociuje k `<portType>` přes atribut `type`, který odpovídá názvu `<portType>`. Vazby specifikuje protokol a podrobné informace o přenosu (například specifikace kódování pro data obsažená v těle SOAP), nicméně, ne skutečné adresy. Vazba se asociuje k jedinému protokolu, v tomto případě HTTP, viz následující:

```
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
```

Nakonec uzel `<service>` seskupí skupinu portů a specifikuje pro ně adresy. Protože v tomto příkladě je použit jediný port, odkazuje se a je provázán s `http://66.28.98.121:9090/soap` pomocí následujícího:

```
<port name="net.xmethods.services.stockquote.StockQuotePort"
binding="tns:net.xmethods.services.stockquote.StockQuoteBinding">
<soap:address location="http://66.28.98.121:9090/soap" />
</port>
```

Za povšimnutí stojí skutečnost, že nic nepropojí SOAP, aby fungoval pouze s HTTP, ani nezapříčiňuje, že se musí vrátit odezvy. SOAP je vytvořen tak, aby byl flexibilním univerzálním protokolem pro zasílání zpráv, přičemž RPC přes HTTP je pouze jednou implementací. Soubor WSDL vám sdělí, jaké služby jsou k dispozici, jak a kde je zpřístupnit. SOAP poté implementuje samotný požadavek a odezvu.

Naštěstí třídy PEAR SOAP zpracují téměř všechno za vás. Chcete-li spustit požadavek SOAP, vytvořte nejprve nový objekt `SOAP_Client` a předejte soubor WSDL službám, které chcete zpřístupnit. Objekt `SOAP_Client` poté vygeneruje veškerý potřebný kód proxy pro požadavky, které se mají provést přímo, alespoň v případě, kde jsou všechny vstupu jednoduché typy Schema. Následující kód představuje kompletní požadavek klienta na demo službu `xmethods.net`:

```
require_once "SOAP/Client.php";
$url = "http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl";
$soapclient = new SOAP_Client($url, true);
$price = $soapclient->getQuote("ibm")->deserializeBody();
print "Current price of IBM is $price\n";
```

`SOAP_Client` provede všechny tajemné věci stojící za tvorbou objektu proxy, který poskytuje přímé vykonání metod specifikovaných v WSDL. Poté, co se provede volání `getQuote()`, výsledek se deserializuje do nativních typů PHP s pomocí `deserializeBody()` a získáte následující:

```
> php delayed-stockquote.php
Current price of IBM is 90.25
```

Přepis system.load na službu SOAP

Rychlý test vašich nových znalostí SOAP spočívá v reimplementaci XML-RPC služby `system.load` na službu SOAP.

Pro začátek definujte službu SOAP jako specializaci `SOAP_Service`. Je třeba implementovat minimálně čtyři funkce:

- **public static function getSOAPServiceNamespace()**{} – musí vracet definovaný jmenný prostor.
- **public static function getSOAPServiceName()** {} – musí vracet název definované služby.
- **public static function getSOAPServiceDescription()** – musí vracet popis řetězce služby, kterou definujete.
- **public static function getWSDLURI()** {} – musí vracet URL, které směřuje na soubor WSDL, kde je popsána služba.

Navíc byste měli definovat všechny metody, které budete volat.

Podívejte se na definici třídy pro novou SOAP implementaci `SystemLoad`:

```
require_once 'SOAP/Server.php';
class ServerHandler_SystemLoad implements SOAP_Service {
public static function getSOAPServiceNamespace()
{ return 'http://example.org/SystemLoad/'; }
public static function getSOAPServiceName()
{ return 'SystemLoadService'; }
public static function getSOAPServiceDescription()
{ return 'Return the one-minute load avergae.'; }
public static function getWSDLURI()
{ return 'http://localhost/soap/tests/SystemLoad.wsdl'; }
public function SystemLoad()
{
$uptime = `uptime`;
if(preg_match("/load averages?: ([\d.]+)/", $uptime, $matches)) {
return array( 'Load' => $matches[1]);
}
}
}
```

Na rozdíl od XML-RPC vaše metody `SOAP_Service` obdrží své argumenty jako normální proměnné PHP. Metoda musí vrátit pouze pole parametrů zprávy odezvy. Jmenné prostory, které si vybíráte, nejsou povinné, ale jsou ověřovány proti specifikovanému souboru WSDL, a proto musí být interně shodné.

Jakmile definujete službu, musíte ji registrovat jako u XML-RPC. V následujícím příkladě vytvoříte nový `SOAP_Server`, vložíte novou službu a nařídíte instanci serveru, aby zpracoval příchozí požadavky: