Katedra informatiky
Fakulta matematiky, fyziky a informatiky
Univerzita Komenského, Bratislava

Python ako jazyk pre vývoj REST webových aplikácií.

Bakalárska práca

Juraj Popovič

Vedúci: RNDr.Peter Rybár, PhD.

Bratislava, 2010

Čestné prehlásenie

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím uvedenej literatúry.

.....

Poďakovanie:

Ďakujem RNDr. Petrovi Rybárovi, PhD. za predstavenie problematiky RESTu, za vysvetlenie mnohých nejasností týkajúcich sa webovej platformy a za ochotné vedenie pri vypracovávaní tejto bakalárskej práce.

Abstrakt

V tejto bakalárskej práci chcem predstaviť problematiku REST-u. Ide o architektonický štýl pre distribuované hypermediálne aplikácie. REST je skratka od Representational State Transfer a bol predstavený v dizertačnej práci Roya Fieldinga v roku 2000. Čerpá z filozofie stavby HTTP protokolu(Fielding je spoluautorom jeho špecifikácie) a nahlas pomenováva princípy, ktoré stoja za úspešnosťou webu. V súčasnosti predstavuje alternatívu voči architektúram založeným na SOAP-e a získava si popularitu pre svoju jednoduchosť, ľahkú implementovateľnosť a mnohé benefity, ktoré prináša.
Python je jazyk, ktorý sa tesí veľkej oblúbenosti medzi vývojármi webu najmä pre svoju priamočiarosť, dobrú čitatelnosť a multiparadigmovosť.
Cieľom tejto bakalárskej práce pre uviesť, zadefinovať a zhodnotiť REST ako architektonický štýl a naprogramovať ukážkovú aplikáciu dodržiavajúcu jeho princípy a demonštrujúcu jeho vlastnosti v jazyku python.

Kľúčové slová: REST, Webservices, Ajax, Python

Obsah

1 Úvod

Web, ktorý pôvodne vznikol ako systém prepojenia distribuovaných vedeckých dát, dnes predstavuje niečo oveľa komplexnejšie. Je to komunikačná, multimediálna, marketingová a integračná platforma. Počas jeho vývoja sa presadili viaceré technológie, ktoré ho posunuli dopredu smerom k čoraz väčšej interaktivite. Časom sa však využitie HTTP protokolu začalo vzďalovať pôvodnej filozofii jeho návrhu a začal byť využívaný ako transportný protokol pre webové služby, ktoré mali poskytovať cez HTTP protokol dosiahnuteľnú funkcionalitu alebo integrovať IT systémy rôznych organizácií. Tým sa však tieto na webe prítomné systémy prestali správať ako súčasť webu - namiesto zdrojov totiž vystavovali špeficické, na diaľku volateľné operácie. Proti tomuto trendu sa ozval niekdajší spoluautor HTTP protokolu Fielding, ktorý vo svojej dizertačnej práci predstavil REST - architektonický štýl, na základe ktorého bol vybudovaný web, prinášajúci benefity, za ktoré vďačí web svojmu rozšíreniu a úspechu. Ten je definovaný kolekciou obmedzení, ktoré sa naňho vzťahujú a z ktorých potom plynú želané benefity. Kľúčový pojem v ňom nie je vzdialená operácia, ale resource - zdroj - konceptuálna entita dosiahnuteľná cez web. Cieľom tejto bakalárskej práce je zadefinovať a popísať tento štýl, ako aj implementovať aplikáciu, ktorá tieto vlastnosti demonštruje. Predchádzať tomu bude krátke pojednanie o histórií, vývoji a komponentoch webu, ktoré popíšem v druhej kapitole. V tretej kapitole, vychádzajúc zo samotnej dizertačnej práce, bude prezentovaný REST ako architektonický štýl spĺňajúci sériu obmedzení. Benefity, ktoré sa s nimi viažu, budú popísané takisto v tejto kapitole. Štvrtá kapitola bude pojednávať o programovacích jazykoch, aby pripravila priestor pre zaradenie jazyku Python. Jeho vlastnosti, populárne webové frameworky a ich vhodnosť pre vývoj RESTovských webových aplikácií bude diskutovaná v piatej kapitole. Posledná, šiesta kapitola bude stručným opisom implementovanej webovej aplikácie, s cieľom ukázať na nej znaky RESTu a vysvetliť, prečo bol na jej realizáciu vybraný webapp framework.

2 WEB

2.1 história

Internet, ako ho poznáme dnes, prechádzal viacerými vývojovými štádiami.

Na úplnom začiatku stáli snahy vôbec prepojiť pocítače a umožniť im vzájomnú komunikáciu. Prvým miľníkom bolo vybudovanie siete ARPANET s pomocou agentúry ARPA (Advanced Research Projects Agency) roku 1969 medzi štvoricou univerzít. Rozvinul sa koncept packetov, samostatných balíkov prenášaných po sieti, do ktorých sa správa rozkladala pri vstupe na sieť a opäť skladala pri príchode adresátovi. Pojmy ako routing alebo switching sa objavili s potrebou zabezpečiť prenos aj v prípade zlyhanie niektorých uzlov siete, keďže v tom období politické napätie mohlo ľahko vyústiť do ozbrojeného konfliktu ohrozujúceho časti siete, ktorá bola od začiatku vedená sčasti ako armádny projekt. Spočiatku ARPANET fungovala na protokole NCP (Network Control Protocol) a FTP(File Transfer Protocol), čoskoro sa však objavila požiadavka širšej štandardizácie, v dôsledku

čoho vznikla pracovná skupina INWG(Inter-Networking Group), ktorá prišla s návrhom protokolu TCP(Transmission-Control Protocol). Ich cieľom bolo vytvorenie systému, ktorý by dokázal spolupracovať s rôznymi typmi špecifických sietí a tvoril by logickú vrstvu nad nimi, nehľadiacu na technologické detaily ich fungovania. Koncom 70-tych rokov tak vznikol TCP/IP protokol. Za rozšírenie vtedy ešte veľmi skromného internetu môže vyvinutie Ethernetu, typu siete, ktorá sa stala viacmenej štandardom pre LAN-y(Local Area Network) v 80-tych rokoch. V roku 1983 uzrel svetlo sveta DNS(Domain Name System), zodpovedný za preklad ľudsky čitateľných adries na ich binárne ekvivalenty.

Na druhej strane zemegule v tom čase, v prostredí ženevského CERN-u(European Organization for Nuclear Research), sa pokladali základy budúceho webu ako konceptu distribuovaných hypermédií. Tim Berners-Lee v potrebe zhromaždovať a triediť rozmanité informácie, poznámky, štatistiky a dáta získané vedcami z celého CERNu počas rôznych experimentov prišiel s myšlienkou hypertextu. Spočiatku sa nestretla s veľkým ohladom, kým v roku 1990 nevymyslel HTTP(HyperText Transfer Protocol) spolu so spôsobom jednoznačnej identifikácie adries dokumentov - pomocou URI(Unique resource indicator). „WorldWideWeb' „ sa tak stal sľubným projektom predstavujúcim prvý prehliadač, ktorý pracoval s hypertextovými odkazmi. Spolu s ním v tom čase vznikli aj HTML príkazy, slúžiace na formátovanie webovského dokumentu.

Do tejto doby boli internet a web v princípe dve rozdielne veci. Internet ako sieť sietí, kde komunikácia dvoch pocítačov prebiehala TCP/IP protokol a Web ako distribuovaná platforma pre zdieľanie štrukturovaných dokumentov. Začiatkom 90-tych rokov webové servery, softvéry schopné odpovedať na HTTP požiadavky webových klientov alebo prehliadačov, začali prenikať do prostredia internetu. Čoskoro sa objavili nové prehliadače, z ktorých najpopulárnejším bol Mosaic a ktorý bol čoskoro nahradený Netscape Navigatorom, za ktorého vývojom stáli v základe tí istí ľudia, ako pri Mosaicu. Neskôr si potenciál webu uvedomili viaceré ako technologické, tak obchodné spoločnosti a to prispelo k rýchlemu rozvoju webových technológií, prehliadačov a ich schopností, ktorý, motivovaný najmä ziskami z internetu, trvá dodnes. Tento prehľad bol založený na informáciách z [3]

2.2 Web, jeho komponenty a vývin

HTTP protokol

Základom celého webu je HTTP protokol, špecifikovaný v [7]. Je to protokol aplikačnej vrstvy v OSI-modele postavený na TCP/IP protokole, ktorý mu zabezpečuje spojenie a prenos samotných správ. Dnes je používaná verzia HTTP 1.1, definovaná v roku 1999 materiálom RFC 2616. Slúži na komunikáciu medzi webových serverom, teda programom, ktorý sprístupňuje dokumenty a služby a webovým klientom(prehliadačom), ktorý o tieto služby v interakcii s používateľom žiada. Základná jednotka komunikácie je správa. Správa od klienta k serveru sa nazýva požiadavka(request), opačným smerom putuje odpoveď(response). Trocha abstraktnejším, ale kľúčovým konceptom je zdroj(resource). Je to akýkoľvek objekt(dokument či služba) nachádzajúca sa na webe, ktorá je jednoznačne identifikovaná svojou URI. Každý resource(budem používať radšej tento pojem namiesto

webového zdroja, ktorý často označuje čosi iné, jazykoví puristi odpustia) má viacero reprezentácií. Pod reprezentáciou potom rozumieme konkrétnu dátovú odpoveď od servera, ktorá sa od iných reprezentácií toho istého resourcu môže líšiť formátom, velkosťou, jazykovou mutáciou, atď.). Tu je dôležité spomenúť MIME typy. Sú to štandardizované formáty, v ktorých je možné dostať odpoveď na požiadavku na daný resource. Medzi typické patria napr.:

- application/javascript
- image/jpeg
- text/html
- text/xml

Každá správa pozostáva z hlavičky a tela. Hlavička charakterizuje dáta požadované alebo prenášané v odpovedi. Hlavička obsahuje riadky formátu parameter:hodnota. Za hlavičkou nasleduje prázdny riadok a potom (nepovinné) telo správy. Requesty aj responsy majú svoje parametre hlavičky, medzi typické parametre requestu patrí

- Host - doménové meno servera
- Accept-Encoding - prípustné kódovania
- Authorization - credentials, teda údaje na potrebné na prihlásenie sa
- If-Modified-Since - direktíva pre zaslanie len dát modifikovaných od uvedeného dáta
- User-Agent - druh klienta, teda napr. webového prehliadača
- Content-Type: - MIME typ požadovaného dokumentu
- Cache-Control - direktívy určujúce správanie sa cacheových mechanizmov voči tejto požiadavke

medzi parametre odpovede patrí

- Content-Encoding - kódovanie poslaného dokumentu
- Expires - definuje obdobie platnosti daného resourcu
- Set-Cookie - nastavuje cookies
- Location - používané pri presmerovaní alebo vytvorení nového resourcu
- WWW-Authenticate - schéma, ktorá sa má použiť pre daný resource

Samotné určenie cieľového resourcu v HTTP requeste by ešte netvorilo zmysluplnú správu. Význam jej dodáva až použitie HTTP metódy. Metóda vyjadruje akciu, ktorá sa má nad daným resourcom vykonať. Metódy sú práve tieto:

- OPTIONS
- GET
- HEAD
- POST
- PUT
- DELETE
- TRACE
- CONNECT

Pre účely tejto práce sú zvlášť dôležité GET, PUT, POST a DELETE, ktoré tvoria tzv. základné CRUD (create, read, update, delete) rozhranie.

Metóda GET by mala vrátiť nejakú reprezentáciu požadovaného resourcu. Jej dôležitou vlastnosťou je *bezpečnosť*. Znamená to, že táto metóda nemá pri správnej implementácií nijak zasahovať do požadovaného resourcu. Naopak metóda PUT slúži na jeho updatovanie, prípadne vytvorenie, ak daný resource ešte neexistuje. DELETE, ako názov napovedá, by mala zmazať daný resource. Všetky tieto metódy navyše zdieľajú vlastnosť *idempotentnosť*. To znamená, že ich viacnásobné zavolanie nad daným resourcom vedie k rovnakému výsledku, ako ich jednorazové volanie, napr. opakované zmazanie daného resourcu je proste zmazanie. Metóda POST slúží na prenos formulárových dát, na posielanie správ, na pridávanie hodnôt do databáz, teda všeobecne na vytváranie resourcov. Je možné ju však použiť takmer lubovoľne a od implementácie webovej aplikácie záleží, ako s ňou naloží. Preto sa používa na prenášanie zložitejších požiadaviek na server, ktoré sa vyskytujú, ako bude neskôr spomenuté, pri webových službách.

Server na požiadavku odpovedá stavovým kódom. Je to číslo zviazané s krátkym popisom vystihujúcim situáciu po vykonaní klientskej požiadavky. Medzi najznámejšie patria:

- 200 OK - požiadavka bola naplnená a súčasťou odpovede sú dáta zodpovedajúce požiadavke
- 201 Created - resource bol vytvorený
- 403 Forbidden - klient nemá práva na vykonanie danej akcie nad daným resourcom
- 404 Not found - resource nenájdený

- 500 Internal server error - na serveri nastala neočakávaná chyba, požiadavka nesplnená

Typická dvojica request - response teda môže vyzerať nejak takto:

request
GET /encrypted-area HTTP/1.1 Host: www.example.com

response
HTTP/1.1 200 OK Date: Mon, 23 May 2005 22:38:34 GMT Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux) Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT Accept-Ranges: bytes Content-Length: 438 Connection: close Content-Type: text/html; charset=UTF-8

Pri klient-server komunikácií vstupujú do hry viaceré komponenty.
PROXY

Proxy je kompoment, ktorý vytvára medzi klientom a serverom ďalšiu medzivrstvu, z pohľadu klienta ma rozhranie ako server a z pohľadu servera naopak ako klient. Ich úlohou je pridávanie ďalšej funkcionality nad správami, ktoré nimi pretekajú, často sa proxy používajú na zachovanie anonymity(skrývajú IP adresu), alebo slúžia na filtrovanie obsahu, ktorý cez ne putuje. Takisto môžu na nich byť umiestnené firewally alebo zariadenia na cachovanie.
GATEWAY
Gateway je taký server, ktorý funguje ako vstupná brána pre nejaký webový priestor, často sú na ňom umiestnené bezpečnotné prvky zamedzujúce v prístupe neželaným programom cez sieť.
WEB CACHE
Cache je mechanizmus umiestňovania už raz stiahnutých dokumentov v lokálnom priestore, aby pri opätovnej požiadavke na ne nebolo potrebné sťahovať čakať na ich získanie zo siete. Nie všetky resourcy môžu byť cacheovateľné, to, či taký daný resource je alebo nie, zavisí na viacerých veciach, mimo iného na metóde, ktorou je volaný a na nastavení HTTP hlavičky pre daný resource.
FIREWALL
Firewall je zariadenie, softvérové(väčšinou) alebo hardvérové, napomáhajúce internetovej bezpečnosti filtrovaním prichádzajúcich a ochádzajúcich správ. Môže chrániť ako individuálny počítač, tak aj celú sieť, keď pôsobí na gateway do nej. Dokáže pracovať na viacerých protokolových úrovniach, napr. na transportnej TCP/IP vrstve alebo aplikačnej HTTP vrstve. V prvom prípade filtruje pakety len na základe ich jednotlivých obsahov, nevie

teda významovo spájať viacero paketov a rozumieť ich obsahu. Používa sa najmä na zamedzenie vstupov na nezvyčajné porty, na zahodenie paketov z od vybraných IP adries a pod. Firewall pracujúci na aplikačnej vrstve toho dokáže viac: na základe HTTP metódy, headrov, obsahu správy a požadovaného URL vie podľa zadefinovaných pravidiel alebo politík rozhodnúť o tom, či bude daná správa preposlaná alebo nie. Tie pravidlá môžu byť buď akceptačné, teda je vymenovaná množina serverom, ktorým sa dôveruje a správy od iných sú zamietané, alebo naopak, existuje množina blokovaných a ostatné sú povolené.

Servery

Web server je softvér(pričom sa zvyčajne počítač, na ktorom tento softvér beží, zvykne označovať server), ktorý, všeobecne povedané, odpovedá na požiadavky klienta. V praxi to znamená najmä poskytovanie dokumentov, najčastejšie webstránok, ale aj vykonávanie rôznej inej funkcionality. Tu treba rozlíšiť dve veci - po prvé program, ktorý manažuje HTTP komunikáciu s klientom a v princípe je schopný odpovedať iba zaslaním správy, príp. dokumentu, a po druhé webovú aplikáciu, ktorá vykonáva nejakú často netriviálnu funkcionalitu. O komunikácií týchto dvoch sa zmienim neskôr. Pôvodný stav, že väčšina funkčnosti, tak ako ju vnímame na webstránkach, sa odohrávala na strane servera a klientovi sa potom posielal často už len statický výstup zobraziteľný v prehliadači, je pomaly nahrádzaný prístupom, keď odpoveď poslaná klientovi obsahuje kód rozširujúci funkcionalitu stránky u klienta - vďaka bohatej interaktivite sa pre to zaviedol pojem RIA - rich internet applications.

CGI a skriptovanie na strane servera

Na vykonávanie funkcionality na strane servera slúžia predovšetkým skriptovacie jazyky(to vyplýva z ich vlastností vhodných pre webový vývoj; aplikačný kód však môže byť napísaný v lubovoľnom inom jazyku, tradičné jazyky ako C++ nevynímajúc), medzi najpoužívanejšie patrí PHP a Python, medzi staršie patrí napr. Perl. Tu si ale treba vysvetliť proces, akým sa požiadavka od klienta premení na parametre nejakej funkcie programu bežiaceho na serveri a naopak ako sa výstup z takéhoto programu dostane klientovi. Tento medzičlánok vypĺňa pojem CGI (Common Gateway Interface), definovaný v [6]. Jednoduché rozhranie CGI káže, že server má zavolať aplikáciu a nastaviť tzv. environment variables, teda akési premenné nachádzajúce sa ma pomedzí servera a programu. Je to napr. QUERY STRING, ktoré má možnosť program prečítať a do ktorej má server uložiť celý reťazec znakov nachádzajúci sa za prvým výskytom „?“ v URL adrese. Klient tento vstup spracuje, vezme si stade čo potrebuje a odpovedá na svoj štandardný výstup. Server prevezme tento výstup a ako správu to pošle klientovi. Je teda na programe, aby vyskladal platnú HTTP odpoveď s hlavičkou aj telom. Mnohé frameworky podporujúce CGI rozhranie toto robia za vývojárov automaticky, pretože na vyskladávanie HTTP odpovedí majú knižnice.

Klienti

Webový klient je akýkoľvek program komunikujúci so serverom prostredníctvom HTTP protokolu. Najčastejšie ide o webové prehliadače, určené na zobrazovanie HTML dokumentov, schopné vykonávať aj istú funkcionalitu, o ktorej bude reč neskôr. Klient však môže byť ľubovoľná aplikácia, ktorá dokáže odpovede zaslané serverom spracovať. Klient môže byť ovládaný človekom, tak ako webové prehliadače, alebo autonómny, teda nejaký program, ktorý automaticky zasiela požiadavky na server a spracúva odpovede. Typicky sa jedná napr. o crawlery či roboty, ktoré slúžia na indexáciu stránok pre vyhľadávače. Keďže vyhľadávanie na webe je jednou z jeho už dnes najpodstatnejších charakteristík, zapadajú webové aplikácie(servery) umožňujúce jednoduché vyhľadávanie lepšie do celkovej architektúry webu.

Webstránka, HTML

Webstránkou sa v základe rozumie HTML dokument, ktorý môže byť obohatený o istú funkcionalitu. HTML je jazyk popisujúci štruktúru dokumentu - teda rozdelenie do rámcov, umiestnenie tabuliek, odkazov, formulárov, centrovanie, zarovnávanie a pod. Druhá vec je vzhľad dokumentu - aj keď HTML umožňuje popisovať do istej miery vzhľad, je vhodné oddelovať kvôli lahšej udržiavatelnosti a variabilnosti štruktúru, zachytenú v HTML tagoch, od štýlu, popísaného pomocou CSS(Cascading Style Sheets). Pomocou nich sa definujú pre jednotlivé štýly fonty, hrúbky písma, farby a pod.

Dynamické webstránky, HTML DOM

Stránky používajúce iba HTML sa ale skoro ukázali ako nedostatočné. Bolo potrebné vymyslieť spôsob, ako vniesť do stránok nejakú jednoduchú funkcionalitu, zabezpečujúcu transformáciu samotného dokumentu, teda pridávanie HTML tagov a textu už počas behu programu u klienta. Odpoveď prišla v podobe skriptovacích jazykov. Medzi prvými sa objavil v roku 1995 javascript, vychádzajúci najmä z jazyka C. Vykonávateľmi týchto skriptov sú samotné prehliadače. Vďaka tomuto sa mnoho výpočtových nárokov ľahko presunulo na klienta. Spočiatku bol využívaný predovšetkým na rôzne dynamické vizuálne efekty. Po uvedení HTML DOM [8], teda Document Object Model-u, však získal hlbší význam. Vďaka nemu vie totiž pristupovať k jednotlivým grafickým HTML prvom ako k objektom, meniť ich funkcionalitu za behu, odstraňovať a pridávať ich, čo umožňuje ďalší level interakcie nezávisle od servera.

Session a cookies

Pri viacerých aplikáciách je potrebné pamätať si niečo o klientoch, ktorí ich práve používajú. Na to slúži mechanizmus sessionov. Session je vlastne nejaký dočasný komunikačný priestor vytvorený medzi serverom a klientom, do ktorého nevidí nikto iný. Na jeho iden-

tifikáciu na strane servera slúží nejaké session ID, jednoznačná charakteristika pridelená jednému takémuto priestoru. Na strane klienta na to slúžia cookies - dáta zachytávajúce stav komunikácie. Cookies sa posielajú od servera spolu s reprezentáciou daného resourcu a prehliadač ich automaticky uloží. Môže to byť napr. spomínané ID danej session, ktorej samotný obsah je uložený v databáze, prípadne môžu cookies obsahovať priamo položky charakterizujúce stav danej komunikácie. Na strane servera sa môžu ukladať informácie o akciách užívateľa, napr. priloženie položky do nákupného košíka, tak, že klient posiela požiadavku s nastaveným cookie, ktoré dostal od servera a ten si len pozrie, ktorej session odpovedá naspäť zaslané session ID a urobí akciu na nej, napr. odošle HTML stránku zobrazujúcu položku už ako pridanú do košíka. Nevýhodou takéhoto prístupu je, že ak daný server zlyhá, celý stav komunikácie medzi serverom a klientom je stratený, navyše sám klient sa nemôže k danému stavu neskôr jednoducho vrátiť, príp. preposlať naňho link niekomu ďalšiemu.

AJAX

Ďalším levelom vo zvyšovaní funkčnosti webstránok je technológia známa ako AJAX(Asynchronous JavaScript and XML), s ktorou sa dá zoznámiť v [9]. Tá umožňuje asynchrónne dotazovanie servera, t.j. žiadosti posielané bez samotného zásahu užívateľa, len na základe programu. Ajax používa *DOM(Document Object Model)* na dynamické zobrazovanie dát a API *XMLHttpRequest* na samotné posielanie a prijímanie HTTP požiadaviek. *XMLHttpRequest* má metódy *open* na vytvorenie HTTP žiadosti(s parametri HTTP metódy, cielovej URL a logickú hodnotu, či chceme poslať správu synchrónne alebo asynchrónne.) a *send* na jej odoslanie. Výhodou je, že browser si nemusí žiadat celý obsah stránky, ale vždy len to, čo potrebuje. Odpoveď dostane buď ako obyčajný text, ale rozšírenejší spôsob je poslať nazad XML alebo JSON súbor, pretože vďaka ich štruktúrovanosti da dajú prenášať celé dátové objekty, ktoré sa následne po nejakej úprave môžu zobraziť na stránke.

XML a JSON

Sú to dva najčastejšie formáty prenosu štrukturovaných dát na webe, pričom JSON sa rozširuje čoraz viac na úkor XML. XML dokument predstavuje vlastne strom, kde každý uzol má svoj otvárací a zatvárací tag ohraničený zátvorkami, pričom uzol môže mať atribúty a jeho telo môže byť buď nejaká vlastná hodnota, alebo ďalší uzol. Týmto je umožnená štruktúrovanosť dát.
Na druhej strane JSON je formát, ktorý je stvorený na predávanie hodnôt medzi rôznymi jazykmi. Podporuje základné a aj agregované dátové typy, nachádzajúce sa vo všetkých moderných jazykoch a preto objekty z nich sú zväčša ľahko prevoditeľné do formátu JSON, najmä ak sa jedná o objektovú reprezentáciu entít, s ktorými pracuje nejaká webstránka. Veľkou výhodou formátu JSON je to, že jeho syntax je priamo súčasťou syntaxe javascriptu a teda objekt vo formáte JSON je priamo prevoditeľný na ekvivalentný javascriptovský objekt, s ktorým sa dá pracovať priamo na strane klienta. Navyše svojím zápisom je JSON

menej „ukecaný“ ako XML, pretože napr. na kolekcie, kde všetky prvky majú v princípe rovnaký typ, nepotrebuje vedieť názov každého jedného elementu, čím sa zmenšujú nároky na dátový prenos, najma ak jeho rozsah zohráva úlohu.
Nasleduje príklad XML vs. JSON na tých istých dátach.

XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<cart>
  <owner>Juraj Popovic</owner>
  <items>
    <item>RESTful webservices<item>
    <item>SOA integration patterns<item>
    <item>Queen II<item>
  </items>
  <amount>1848.48</amount>
</cart>
```

JSON:

```
takto moze vyzerat JSON-string:

{"owner":"Juraj Popovic","amount":"1818.48",
  "items":["RESTful webservices","SOA integration patterns","Queen II"]}

a takto sa to transformuje do javascript objektu:

val cart = JSON.parse(jsonstring);
```

Viac k nim nájde človek na [11] a [14].
Takto môže vyzerať jednoduchá stránka postavená na ajaxe, ktorá si dokáže vypýtať určité dáta bez potreby uploadovania stránky:

Po kliknutí užívateľa na button sa zavolá funkcia getData, ktorá vytvorí AJAXový request a pošle ho. Pri tom sa definuje funkcia(mohla by sa definovať aj inde), ktorá sa ma postarať o odpoveď. Tá musí skontrolovať, či je odpoveď už prijatá(readyState==4) a či bolo volanie úspešné(HTTP status==200; vo všeobecnosti môže byť aj iný, napr. 301 pre vytvorenie dokumentu a pod.) a podľa toho sa rozhodne buď užívateľa upozorniť, alebo prijaté dáta zobraziť.

```
<html>
<head>
<script>

function RenderDataOnPage(responseText) {
// zobrazi ziskane data na stranke
}

function getData()
{
    var xhr;
    try {  xhr = new ActiveXObject('Msxml2.XMLHTTP');   }
    catch (e)
    {
        try {   xhr = new ActiveXObject('Microsoft.XMLHTTP');    }
        catch (e2)
        {
          try {  xhr = new XMLHttpRequest();     }
          catch (e3) {  xhr = false;   }
        }
     }

    xhr.onreadystatechange  = function()
    {
         if(xhr.readyState  == 4)
         {
              if(xhr.status  == 200)
                  RenderDataOnPage(xhr.responseText);
              else
                 alert("Error code " + xhr.status);
         }
    };

   xhr.open("GET", "/repository/data",  true);
   xhr.send(null);
}
</script>
</head>

<body>
...
  <button onClick = "getData();">show me data!</>
...
 </body>
 </html>
```

WEB 2.0

Web 2.0 nie je presný technický pojem, ale označuje množinu technológií a prístupov použitých k tvorbe nového typu webu - multimediálneho, dynamického, kde užívatelia spolupracujú, zdielajú informácie a multimédiá, vytvárajú sociálne siete, formujú tematické wiki atď. Ako hovorí Tim O´Reilly, otec pojmu Web 2.0:

„ Web 2.0 is the business revolution in the computer industry caused by the move to the Internet as a platform, and an attempt to understand the rules for success on that new platform!“

Hlavná zmena oproti starému webu je filozofia, pohľad na web. Namiesto pasívneho prostredkovávania informácií by mal byť platformou na ešte lepšiu a prepojenejšiu komunikáciu užívateľov s možnosťou aktívnej tvorby obsahu. Na web 2.0 sa dá pozerať z rôznych uhlov pohľadu. Pre biznismenov znamená nové možnosti prilákania zákazníkov, propagácie tovaru a služieb, budovanie vlastných zákazníckych sietí, nové možnosti starostlivosti o zákazníka, prepojený marketing atď. Zo sociologického hľadiska sa takýto web stáva novou komunikačnou platformou, majúcou vlastné pravidlá a meniaciou spoločnosť. To je vidieť najmä na úspechu sociálnych sietí typu Facebook, ktoré pomaly, ale iste transformujú spoločnosť a prinášajú nové typy vzťahov, hodnôt, prepojení. Z nášho hľadiska je teraz ale zaujímavé vnímať web 2.0 ako ihrisko nových technologických konceptov, resp. renesanciu starých, medzi ktoré patrí aj REST. Ten je totiž v princípe jednou z dvoch odlišných možností, ako môžu spolu automaticky komunikovať dva vzdialené počítače cez internet, a takáto automatizovaná komunikácia je jedným z pilierov Webu 2.0, popri Ajaxe. Doležité je práve to, aby aplikácie poskytujúce priestor na tvorbu užívateľmi zadaného obsahu boli v pravom zmysle slova súčasťou webu, aby entiny, dáta, zdroje, ktoré sa vytvoria boli jednoducho dostupné pre ostatných práve tak, ako sú dostupné ľubovoľné webstránky. Práve toto umožňuje prístup založený na RESTe, ako neskôr vyplynie z jeho vlastností.

2.3 Webové služby

S vývojom webu sa zrodil aj koncept webových služieb - funkčných jednotiek distribuovaných po internete, ktoré poskytujú užívateľom, ale aj iným webovým službám svoje služby. Základnou komunikačnou jednotkou bola vzdialená operácia. Otázkou bolo, ako volať po internete funkcionalitu danej služby a ako sa dozvedieť, čo všetko poskytuje.

Distribuovaná komunikácia

Potreba komunikácie rôznych softvérových komponentov, či už v rámci jednej organizácie alebo naprieč hranicami firiem viedla už dávnejšie k vývoju rôznych riešení. Naznámejšími boli proprietárny model DCOM(Distributed component object model) firmy Microsoft, ktorý sa opieral o technológiu COM a CORBA(Common Object Request Broker Architecture), ktorá normalizovala sémantiku volaní medzi funkciami napísanými v rôznych programátorských jazykoch a používala IDL(interface definition language) na popis

jednotlivých takýchto rozhraní. Neskôr sa však ukázalo, že takéto volania neprechádzajú cez firewally a zvíťazil flexibilnejší prístup založený na prenose funkčných volaní cez HTTP protokol.

XML-RPC

Alebo Remote Procedure Call. Je to všeobecný názov pre prístup založený na priamom invokovaní vzdialených procedúr. Ako myšlienka sa to objavilo prvýkrát v roku 1976, popisom v RFC 707. Keďže rôzne platformy potrebujú rozdielny spôsob komunikácie, bol vyvinutý IDL(Interface description language) na špecifikovanie rozhraní tak, aby spolu mohli komunikovať aj programy napísané v róznych jazykoch a bežiace pod róznymi systémami. Novšou variantou je cez HTTP POST realizovaný XML-RPC, ktorý volá procedúry tak, že ich názvy a parametre zabalí do XML dokumentu, ktorý posiela serveru.

SOAP

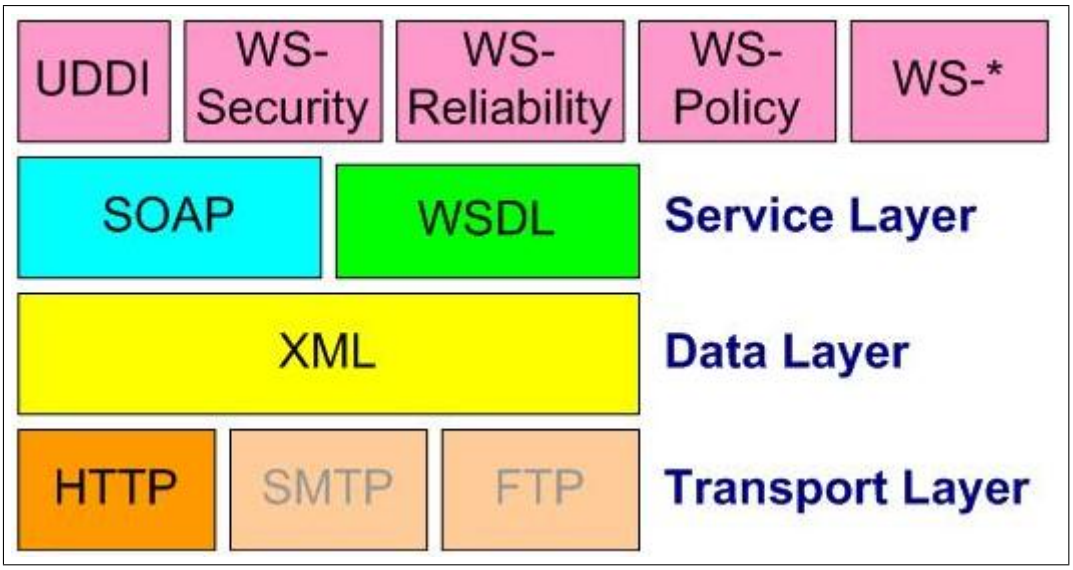
Alebo Simple Object Access Protocol, popísaný na [10]. Je to protokol slúžiaci na implementáciu web servisov, fungujúci cez RPC alebo HTTP, používajúci na prenos správ a dát XML. Je jazykovo a platformovo nezávislý. Práve extenzívne používanie HTTP ako prenosového protokolu nesúceho SOAPovské XML správy viedlo k tomu, že pôvodní tvorcovia HTTP špecifikácie obviňovali propagátorov SOAPu z nepochopenia filozofie HTTP, v dôsledku čoho vznikol aj samotný REST. SOAP všetky svoje volania, nezávisle od ich sémantiky, realizuje prostredníctvom metódy POST, čím úplne obchádza filozofiu stavby HTTP prokotolu. Krátky príklad soapovskej správy:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

SOA

S čoraz väčšou integráciou IT do biznis procesov sa zmenil pohľad na spôsob, akým by systémy mali spolu komunikovať. Na splnenie čoraz komplexnejších požiadaviek už nestačilo mať jednotlivé navzájom sa volajúce a nejakú úzku funkcionalitu vykonávajúce softvérové komponenty. Bolo potrebné vymyslieť architektúru, ktorá by umožňovala komunikovať ľubovoľným dvom systémom prepojeným cez sieť, a ktorá by brala spĺňala nároky na bezpečnosť, nezávislosť systémov a univerzálnosť. Tak vznikol pojem servisne orientovanej architektúry. SOA predstavuje prístup k tvorbe architektúry založený na koncepte služieb. Celá architektúra distribuovaného systému by tak mala vyzerať ako spleť služieb, ktoré sa navzájom využívajú a ktorých operácie sú popísané cez WSDL(Web service description language). Každá služba má svoj endpoint, teda nejakú svoje umiestnenie na webe (URL). Vďaka WSDL jazyku vie popísať poskytované metódy, ich vstupné parametre a návratové hodnoty. Výhodou oproti komponentovému prístupu je nízka previazanosť (loose coupling) jednotlivých súčastí, čo je opodstatnená požiadavka, pretože tie sa môžu často meniť a zmena je tým jednoduchšia, cím menej vzájomných prepojení a závislostí v systéme existuje. SOA používa middleware v podobe ESB(Enterprise service bus), ktorý sa stará o samotný mechanizmus doručovania správ medzi jednotlivými komponentami. Servisne orientované architektúry ako biznis koncept používajú v súčasnosti najčastejšie protokol SOAP. Pri SOA bolo treba zadefinovať mnoho štandardov, aby sa zabezpečila interoperability, bezpečnosť a pod., takže sa postupne vyvinula celá množina týchto štandardov,

ktorá ale sťažuje implementáciu. Viac napríklad na [5]. Tento obrázok zachytáva koncept SOA:

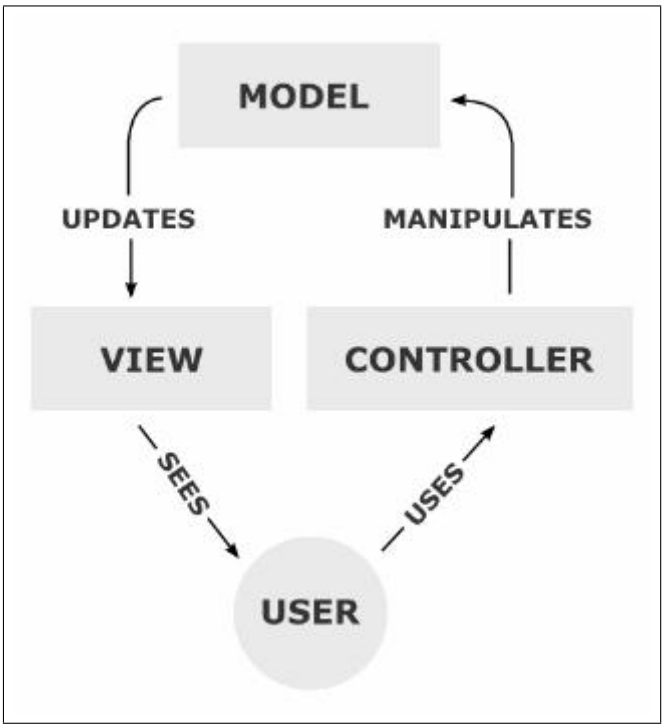


2.4 Vývoj webových aplikácií

MVC

Architektonický vzor Model-View-Controller je pre webové aplikácie typický. Často sú na jeho filozofii založené webové frameworky. Oddeľuje grafickú vrstvu, dátovú vrstvu a oblužnú vrstvu. Dátová vrstva sa stará o samotnú vnútornú logiku aplikácie, o narábanie s dátami, ich validáciu a transformáciu. V tejto vrstve sa implementujú biznis pravidlá. Model môže predstavovať napr. stav užívateľského účtu, objednávku a pod. Oblužná vrstva zabezpečuje prenos požiadaviek správnym metódam a volá funkcie dátovej vrstvy. Je zodpovedná za fungovanie aplikácie v infraštruktúrnom zmysle, nestará sa o sémantiku dát, ktorých prenos zabezpečuje. Grafická vrstva sa stará o tvorbu konečného grafického výstupu zobrazujúceho vhodným spôsobom výstupy dátovej vrstvy. Tvorí užívateľské rozhranie, pomocou ktorého môže človek s aplikáciu komunikovať. Ten istý model môže mať kľudne viacero grafických výstupov. Prísne oddelenie zodpovednosti za jednotlivé časti prispieva lepšej kontrole nad aplikáciou a možnosť nezávislého vývoja jednotlivých komponentov.

Nasledujúci obrázok koncepčne zachytáva interakcie v rámci tohto vzoru.



Webové frameworky Webové frameworky sú ucelené softvérové balíky, ktoré slúžia na zefektívnenie práce na vývoji webových aplikácií a zastrešujú typickú, opakujúcu sa funkcionalitu, ktorá je pre väčšinu web aplikácií spoločná. Vyvinuli sa ako ďalší stupeň knižníc poskytujúcich jednotlivé funkčné celky. Existujú prakticky pre každý jazyk používaný na web development. V princípe sa delia na dva základné typy, request a component based.

Request based

Ich architektúra je navrhnutá tak, že očakáva vstupy, requesty, ktoré spracuje a na ktoré potom pošle odpoveď. Opiera sa teda prirodzene o stavu a filozofiu webu. Ich starosť spočíva v parsovaní URL, mapovaní URL na oblužné metódy a preposlanie výsledného HTML dokumentu.

Component based

Frameworky založené na komponentoch sa na druhej strane snažia budovať abstrakciu nad HTTP protokolom a miesto request-responsu pohľadu sa na webovú aplikáciu dívajú ako na sadu komponentov a HTTP komunikáciu zaobalujú ako interakciu medzi komponentami. Snaha je najmä priblížiť vývoj webu vývoju desktopových aplikácií, čím ale popierajú princípy, na kt. je web postavený. Kvôli tomuto netransparentnému prístupu, keď stav klient-server interakcie je zachytený v sessione na serveri, sú tieto frameworky pre REST nepoužiteľné.
Medzi užitocnú funkcionalitu frameworkov patrí napr. ľahké vytváranie dátových mode-

lov a zaobalenie práce s databázou, podpora a využitie sessionov, URL routing, mapovanie URL na metódy programu, správa bezpečnosti pomocou ľahkej autorizácie a autentifikácie, podpora cacheovania, jednoduchá práca s HTTP headermi, objektový prístup k requestu a responsu alebo templatovací systém, vďaka ktorému je možné jednoducho dopĺňať statický HTML kód podla potrieb programu v runtime. Výhodnými sú potom také templatovacie systémy, v ktorých existuje pojem dedenia. Podpora AJAXu takisto patrí medzi želateľné vlastnosti. Architektúra frameworkov môže, ale nemusí implementovať vzor MVC. Tie, ktoré tak robia, sú pre vývoj REST aplikácií vhodnejšie, pretože vyhovujú jeho filozofii, kde kontroller sa stará o mapovanie HTTP metód, modely predstavujú jednotlivé resourcy a view vrstva je zodpovedná za tvorbu požadovaných reprezentácií.

3 REST

Písal sa rok 2000, keď boli v dizertačnej práci amerického počítačového vedca Fieldinga sformulované jeho predstavy o architektúre distribuovaných aplikácií. Vychádzajúc z filozofie HTTP protokolu zadefinoval architektonický štýl, ktorý využíva prednosti HTTP protokolu v ďaleko plnšej miere, než napr. SOAP. V princípe išlo o explicitné pomenovanie tých existujúcich vlastností webu, ktoré mu zaručili úspech. Za 9 rokov odvtedy sa tento štýl dostal do povedomia internetovej a softvérovej komunity a vedú sa mnohé spory o jeho použitelnosti a vhodnosti pre jednotlivé prípady. Isté však je, že svojimi myšlienkami prispel k formovaniu budúceho webu, pretože slovné spojenia ako „RESTovská aplikácia“ alebo „RESTovské webové služby“ sa dnes už bežne používajú na označenie čohosi pokrokového. Aj keď nie vždy oprávnene.

3.1 Architektúra aplikácie, architektonický štýl

Skôr ako sa začnem venovať RESTu samotnému, je potrebné vymedziť si niektoré pojmy, aby sa REST ako koncept správne chápal. Architektúrou softvéru rozumieme abstraktnú logickú štruktúru a správanie sa jeho častí, pričom nás zvlášť zaujímajú ich vzájomné pôsobenia a rozhrania. Definícií architektúry je snaď toľko čo architektov samotných, takže snaha o nájdenie nejakej jednotnej, vše zahŕňajúcej definície ja zbytočná. Predsa však uvediem niekoľko ilustratívnych citácií, ktoré sa uvádzajú v [18]:

Hayes-Roth, 1994:
„...an abstract system specification consisting primarily of functional components described in terms of their behaviors and interfaces and component-component interconnections. “

Bass, et al., 1994:
„...the architectural design of a system can be described from (at least) three perspectives – functional partitioning of its domain of interest, its structure, and the allocation of domain function to that structure “

Garlan and Perry, 1995:

„The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time. “

Pod architektonickým stýlom potom rozumieme množinu štrukturálnych a funkčných atribútov, ktoré majú architektúry patriace do toho istého štýlu rovnaké. Štýl sa nepozerá na konkrétne charakteristiky jednotlivých architektúr a spôsob ich realizácie, ale určuje len najviditelnejšie znaky z hľadiska štruktúry, ktoré by sa mali dodržať. REST nie je softvér, nie je to štandard, nie je to protokol, nie je to architektúra, je to architektonický štýl. Ten chápe Fielding nasledovne:

Än architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those components within any architecture that conforms to that style. "

3.2 formálna definícia RESTu

Fielding sa teda architektonické štýly rozhodol definovať podľa obmedzení, ktoré sa na ne vzťahujú. Myšlienka začína tzv. nulovým štýlom, kt. nemá žiadne obmedzenia. Každé obmedzenie so sebou ale prináša isté vlasnosti, ktorými daná aplikácia potom disponuje. Medzi tie najdôležitejšie, ktoré môže sieťová aplikácia nadobudnúť, patria:

V nasledujúcom textu budeme komponenty chápat ako entity spracovávajúce, transformujúce dáta, ktoré fungujú prostredníctvom konektorov, abstraktných entít zodpovedných za komunikáciu medzi komponentami. Tento text je založený na [18]

Želatelné vlastnosti distribuovaných aplikácií:

Výkon Existuje viacero pohladov na výkon. Dá sa merať ako priepustnosť, teda miera toho, aký objem dát vie prejsť medzi komponentami za jednotku času. Z hľadiska užívatela má však väčší význam merať napr. latenciu, teda čas, za ktorý dôjde k viditelnej odozve, alebo čas dokoncenia, teda dobu, za ktorú sa preniesie kompletná informácia. Tieto metriky stoja v nepriamej úmere - ak chcem, aby sa užívatelovi zobrazovali priebežne zmeny, stojí to čas navyse, ktorý by sa inak doprial spracovávaniu prichádzajúcich dát. Pre konkrétne prípady potom treba zhodnotiť, ktorá z nich je dôležitejšia. Pre ludí bude zrejme interaktivita klúčovou, kým pre webové crawlery je podstatné rýchle získanie kompletnej informácie. Hoci architektonický štýl nemôže prekonat fyzické ci infraštruktúrne obmedzenia a preto napr. zlepšit cas prenosu daného množstva informácií, môže vhodným spôsobom návrhu prispiet k minimálnej interakcií komponentov.

Škálovatelnosť Tá predstavuje schopnost systému jednoducho, ideálne pocas behu pripájat dalšie komponenty schopné spracovávat žiadosti a pokrývať tak narastajúci rozsah zátaže. To sa dá dosiahnuť decentralizáciou, jednoduchosťou komponentov, správnou voľbou umiestnenia stavu aplikácie a nízkou previazanosťou komponentov. Sessiony držané na serveri môžu napr. bránit efektívnej škálovatelnosti, lebo si nemôžu lahko odovzdávat

rozrobené požiadavky, kedže vnútorný stav daného procesu je známy iba serveru, ktorý s klientom práve komunikuje.

Jednoduchosť Hoci sa táto vlastnost tažko definuje explicitne, je podstatná, pretože uľahčuje samotnú implementáciu aplikácie, umožnuje lahšie pochopit jej architektúru a predovšetkým prispieva k dobrej využitelnosti danej aplikácie inými systémami. RESTovské aplikácie sa mimo iného vyznacujú práve výraznou jednoduchosťou a nízkymi voprednými znalosťami potrebnými na pracovanie s ňou.

Modifikovatelnosť Táto patrí k najpodstatnejším architektonickým vlastnostiam aplikácií všeobecne. Požiadavky na systém na čoraz častejšie menia a preto systémy, ktorým architektúra nedovolí adekvátne sa adaptovať, sú odsúdené na neúspech a nahradenie inými. Pritom nejde len o to, že si zadávatelia systému primyslia novú funkcionalitu, ale aj o to, že jednotlivé podsystémy môžu byť v správe rôznych organizácií, kde každá môže mat odlišné potreby na dalšie rozširovanie systému. Ten sa môže menit na viacerých úrovniach. Konfigurovatelnost znamená, že konkrétne správanie sa systému je inštanciou všeobecnejšieho modelu, kde jednotlivé detaily sú parametrizovatelné. Kastomizovatelnosť hovorí o schopnosti jednotlivých komponentov meniť svoje vnútorné fungovanie bez toho, aby sa tým ovplyvnila funkčnosť ostatných. Rozšíritelnost poskytuje systému nové funkcné možnosti a negatívne závisí od väzieb medzi komponentami. Vyvíjatelnosť predstavuje možnost systému absorbovať nové požiadavky naň tak, aby zostala jeho interakcia s ostatnými konzistentná. Znovupoužitelnosť sa dá dosiahnuť dostatočnou všeobecnosťou komponentových rozhraní.

Transparentnosť Táto vlastnosť hovorí o schopnosti daného komponentu nahliadnuť alebo ovplyvniť stav komunikácie medzi inými. Čím transparentnejšia je architektúra, tým výkonnejšia môže byt, pretože sa dá lahšie distribuovať zátaž.

Spolahlivosť Spolahlivé sú také systémy, ktoré dokážu spracovat nepredvídané chybové stavy bezpečne a konzistentne. K tejto vlastnosti napomáha transparentnosť.

Adresujúc tieto vlasnosti, popisuje Fielding množinu rôznych architektonických štýlov možných pre distribuované aplikácie.

Uvediem tu však len tie, ktorých kompozíciou sa odvodil štýl REST.

Dátové prúdy Známe najmä z unixových systémov sú rúry a filtre, koncept spočívajúci v požiadavke nulového previazanie komponentov, teda jednotlivých rúr. Každá rúra je nezávislá od dalších, nielenže neudržiava žiaden stav komunikácie, ale v princípe ani nevie, kto sa nachádza na jednotlivých stranách jej rozhrania. Slúži teda výlučne na transormáciu dát. Takéto rúry sa musia vedieť skladať za sebou, vytvárajúc tak zloženú transformáciu, co zjednodušuje chápanie systému. Priamo z definície tohto štýlu vyplýva využitelnosť jednotlivých komponentov. Rozšíritelnosť je zrejmá z toho, že pri pridávaní dalšieho filtra sa nemusia ostatné vôbec meniť, podobne ako v prípade jeho vnútornej obmeny. Medzi

nevýhody potom patrí nízka interaktivita a tiež zbytocný čas cakania, ak systém nepodporuje dávkové spracovanie. Požiadavkou navyše môže byt univerzálnosť rozhraní. Práve tá zohráva klúcovú úlohu aj pri RESTe.

Replikované úložište Podstata tohto štýlu spočíva v tom, že dáta sú inštanciované, teda replikované na viacerých miestach, čo umožnuje lepšiu dostupnosť napr. v prípade výpadku jednotlivých serverov. Voči užívatelovi však tento distribuovaný systém vystupuje ako jeden komponent. Najväcšou starosťou pri tomto type štýlu býva konzistentnosť - zaručiť, že zmena na dátach sa prejaví na všetkých ich inštanciách. Odvodeným štýlom je potom cache (budem používat anglické písanie), ktorá uchováva odpovede na jednotlivé requesty. Vylepšená cache si môže pripraviť vopred odpovede na predpokladané dopyty. Pri HTTP protokole je cacheovatelnosť jednotlivých zdrojov nastaviteľná pomocou header parametrov. Tento štýl podporuje predovšetkým výkon aplikácie, pretože znižuje mieru komunikácie smerom ku konečným serverom. Oproti klasickému replikovanému úložistu má tú výhodu, že netreba duplikovať dáta, ktoré sa nevyžadujú.

Hierarchické štýly Medzi najtypickejší štýl patrí klient-server. Ide o situáciu, kde jeden komponent (server) čaká na pripojenie iného(klienta), ktorý posiela requesty, ktoré server spracúva. Aktívna entita je teda klient. Tento štýl je dobrý na rozdelenie zodpovednosti, kde úlohou servera je zväcša manipulácia nad dátami a úlohou klienta je sprostredkúvanie užívateľského rozhrania, čím sa prispieva ku škálovatelnosti. Samotným týmto rozdelením kompetencií sa ešte ale neurčuje, akou technológiou budú jednotlivé volania klient-server prebiehať. Dalším štýlom z tejto kategórie sú vrstvené systémy, kde jedna vrstva poskytuje služby vyššej a využíva služby nižšej vrstvy. Napomáha to znovuvyužitelnosti a vdaka nezávislosti tried od seba vzdialených aj vyvíjatelnosti, na druhej strane zvyšuje celkový sieťový traffic potrebný na prenos daných údajov kvôli medzivrstvovej komunikácií a nabalovaniu správy o kontrolné atribúty protokolov dalších vrstiev, tak ako je to v modele OSI. Spojenie klient-server a vrstveného systému vo webovom priestore predstavujú pridané medzikompontenty ako gatewaye a proxy. Dalším významným obmedzením môže byť požiadavka bezstavovosti. To znamená, že servery neuchovávajú žiadnu formu sessionov, a celý stav komunikácie musí byt uchovávaný na strane klienta a teda prenášaný pri každom requeste. Hoci to môže negatívne vplývať na výkon, výrazne to zlepšuje hned štyri vlastnosti naraz: jednoduchosť, lebo bezstavový server na implementuje lahšie, škálovatelnosť, vďaka tomu, že server môže po vybavení požiadavky uvolniť zdroje s ňou súvisiace a nasledujúci request v rámci tej istej transkacie môže už vykonat iný server, transparentnosť, pretože kompletný stav danej transakcie sa dá vytiahnuť zo samotného requestu a spolahlivosť preto, že vďaka tomu vieme ľahko vykonať nápravné operácie v prípade zlyhania. Kombináciou týchto všetkých vzniká vrstvený, cacheovatelný, bezstavový klient-server štýl.

Kód na požiadanie Spôsob, akým urýchlit užívatelom vnímaný výkon je ak myšlienka prenosu kódu. Namiesto samotných dát pre každý request sa prenáša kód schopný tento request spracovat na mieste volania. Komunikácia medzi komponentami nachádzajúcimi sa

v tom istom procesnom priestore(napr. na jednom počítači) prebieha oveľa rýchlejšie ako taká realizovaná po sieti. Zmysel to má najmä v prípadoch, ked má server vykonat nejaké transformácie klientom zaslaných dát, ktoré ale nevyžadujú prístup k dalším resourcom na serveri a teda sú realizovatelné aj na strane klienta. Príkladom môže byt javascriptová dynamická manipulácia s obsahom stránky namiesto generovania vždy nového HTML súboru na serveri na základe užívateľovej interakcie.

REST ako kompozícia architektonických štýlov Formálnou definíciou RESTu je vlastne kompozícia daných štýlov. Dedí pritom všetky ich spomenuté vlastnosti. Vyznačuje sa teda jednoduchosťou a nezávislou vyvíjatelnosťou vďaka uniformnému rozhraniu, transparentnosťou a spolahlivosťou najmä vďaka bezstavovosti, škálovatelnosťou vďaka bezstavovosti a rozdeleniu na hierarchické štruktúry, rozdelením zodpovednosti vďaka diferencácie klientskej a serverovej časti, výkonom vďaka cacheovatelnosti, modifikovatelnosťou skrz kód na požiadanie. Presné vymedzenie RESTu nájde človek v [18].

Nasledujúci obrázok graficky znázorňuje odvodenie RESTu
(RR=Replicated Repository, $= cache, CS = client server, CSS = stateless CS, LS=layered, COD=code on demand, U= uniform interface)

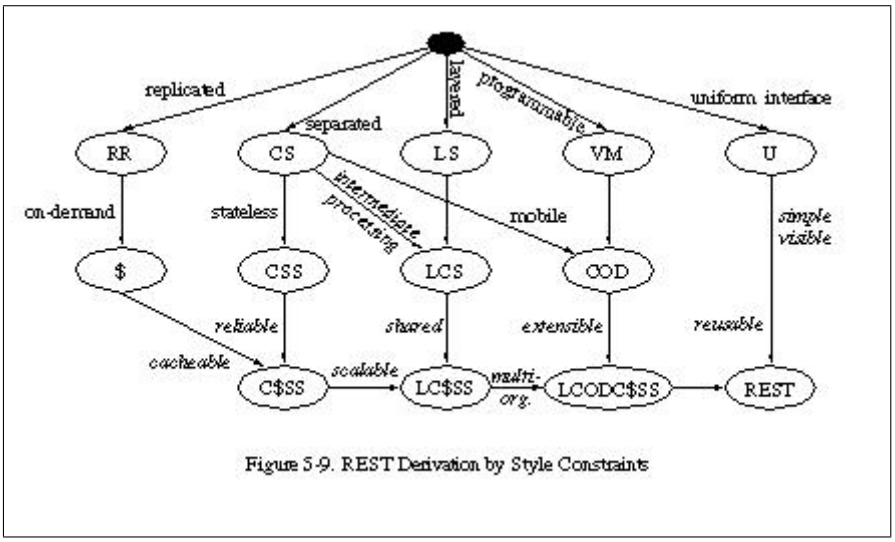


Figure 5-9. REST Derivation by Style Constraints

3.3 praktická definícia RESTu

Predošlý text zadefinoval REST z teoretického hladiska, vychádzajúc z metodiky použitej vo Fieldingovej práci. Hoci REST nie je nutne viazaný na žiaden konkrétny protokol, jeho nasadenie je prirodzene najčastejšie v prostredí internetu cez HTTP protokol, kde samotný web stelesňuje jednu veľkú implementáciu RESTovských princípov. Nasledujúci text predstaví REST z praktického hľadiska, v termínoch zrozumiteľných webovým vývojárom. Vychádzal som pritom najmä z [1] a z [17].

Identifikácia zdrojov Kľúčový pojem je v tomto koncepte zdroj. Zdroj je pre aplikáciu akákoľvek zmysluplná entita, ktorá môže byť pre používateľov zaujímavá. Môže to byť statická, alebo dynamicky sa meniaca entita, ktorej hodnota sa mení s časom. Zdroj môže zodpovedať "fyzickému"zdroju, teda napr. súboru, alebo pamäťovému záznamu, a to či už jednoduchému záznamu, ako napr. telefónnemu číslu človeka, alebo kompozícií záznamov, ako napr. kontakt na človeka(adresa, telefón, email,..). Zdroje nie sú len samotné entity, ale aj kolekcie týchto entít. Pre internetové kníhkupectvo napr. by teda zdroje mohli byť užívateľ, kniha, sada kníh danej kategórie, čo sú viacmenej statické entity, ale aj napr. stav nákupného košíka daného užívateľa v danom čase, čo je dynamicky sa meniaca entita.

REST hovorí, že zdroje musia byť identifikované pomocou jednoznačného URI(uniform resource identifier). Ten môže mať teoreticky hociaký vývojárom vymyslený formát či syntax, pokiaľ dodržiava princíp jednoznačnosti a dané URI ukazuje vždy na tú istú entitu. Dobrým zvykom je vychádzať z konceptu URL a teda stavať URL podľa toho, ako sú logicky entity štrukturované. Pod *www.shop.sk/books* by teda bolo možné nájsť kolekciu kníh, pod *www.shop.sk/books/science* kolekciu kníh o vede a pod *www.shop.sk/books/science/History-of-mathematics* konkrétnu knihu.

Implementácia uniformného rozhrania Kľúčovou vlastnosťou RESTu je, že sa zamierava na podstatné mená namiesto slovies. A teda namiesto toho, aby webová aplikácia definovala sadu funkcií, ktorú je schopná vykonať, definuje sadu zdrojov, nad ktorými je schopná vykonávať (nie nutne všetky) elementárne CRUD operácie - create, read, update, delete. Žiadne iné operácie nie sú dovolené. Zároveň tieto operácie sú dostatočné všeobecné na to, aby dokázali pokryť akúkoľvek operáciu nad daným resourcom, lebo vskutku nič viac než vytvoriť, updatovať, čítať a zmazať daný zdroj človek nemôže chcieť. Úloha architekta systému potom spočíva v tom, aby dokázal premeniť tradičné procedurálne myslenie na zdrojovo-orientované. Toto jednotné CRUD rozhranie je v HTTP protokole prirodzene mapované na metódy GET(create), GET(read), PUT(update), DELETE(delete). Skutočne web ako sada prelinkovaných stránok funguje na tomto princíp. Potrebné je ale dokázať zovšeobecniť chápanie zdroja - tým nemusí byť iba webstránka, ale akákoľvej zmysluplná entita, ktorá môže byť výsledkom nejakej operácie nad dátami na serveri. Spomenuté rozhranie má ešte svoje sémantické obmedzenia. Podobne ako v HTTP, musí byť GET metóda bezpečná, PUT a DELETE idempotentné a odpoveďou na POST request by mala byť(nevynucuje si to priamo štýl, ale patrí to k best practices) buť reprezentácia nového zdroja alebo link na ňu.

Reprezentácie zdrojov REST hovorí, že zdroj je len konceptuálna entita, nie "fyzická". Analogicky ako v Platónovej filozofii, človek nemôže pracovať priamo s danou entitou(napr. dostať ju príkazom GET), ale len prostredníctvom tzv. reprezentácií. Reprezentácia je teda akési pomyslené mapovanie konceptu na konkrétny virtuálny dátový "nosič". Entita kniha môže byť teda reprezentovaná ako webstránka (html súbor) s informáciami o nej, ako xml alebo json dokument popisujúci ju, ako jpeg obrázok zachytávajúci jej podobu, atď. Podstatné je, že reprezentácie by mali byť v najvyššej možnej miere štandardné, pretože tak

budú zrozumiteľné pre väčšinu klientov, napr. všetky moderné webové prehliadače dokážu spracovať html, ale aj xml, jpeg, gif, txt a iné súbory. Daná entita môže byť mapovaná do rôznych reprezentácií súčasne a malo by byť na klientovi, ktorú si rozhodne si vyžiadať, čo môže nastaviť pomocou Accept-Type parametru HTTP headera. Pre jednotlivé reprezentácie je potom rozumné nastaviť cacheovetelnosť pre zvýšenie výkonu aplikácie. Podstatné je uvedomiť si, že vďaka tejto variabilite formátov sa dajú implementovať rôzni klienti a tak k danej aplikácií bude môcť pristupovať aj špecificky napísaný klient schopný spracovávať xml. To môže mať zmysel, ak má klient pristupovať napr. k dátam na lokálnom počítači(čo webový server v princípe pre špefické potreby nevie), pre lepšie užívateľské rozhranie, pre použitie rýchlejšieho/obľúbeného jazyka, atď.

Samopopisnosť a prelinkovanosť zdrojov To, čo robí web webom, je vzájomná prepojenosť webstránok. Tá umožňuje to, na čo sa vyvinul slang "surfovanie". Táto nanajvýš užitočná vlastnosť, robiaca z webu skutočnú informačnú pavučinu, je požadovaná aj RESTom. Reprezentácia by teda okrem informácií o entite samotnej mala niesť aj linky na entity s ňou súvisiace(a to ako v danej aplikácií, tak aj mimo nej), pretože žiadna entita neexistuje izolovane od iných. Tak napr. html dokument reprezentujúci knihu by mohol obsahovať linky na kategóriu, do ktorej patrí, na informácie o autorovi(v rámci kníhkupectva alebo to môže byť odkaz na wikipedii), atď. Ďalšou požiadavkou je samopopisnosť, aby z reprezentácie samotnej bolo jasné, o čo sa jedná, a takisto kvôli správnemu automatickému handlovaniu, aký formát má, to sa nastaví pomocou Content-Type headera.

Hypermédiá ako nositeľ stavu aplikácie Poslednou požiadavkou je neprítomnosť sessionov na strane servera. Nositeľmi stavu komunikácie, transakcií, sú hypermédiá, reprezentácie. To znamená, že daná pokiaľ reprezentácia predstavuje nejaký krok vo viackrokovej transkacii medzi serverom a klientom, musí obsahovať všetky informácie potrebné na posun k dalšiemu, príp. vráteniu sa k predchádzajúcemu kroku. V princípe musí byť každý ďalší request realizovateľný iným serverom, ktorý nemá potuchy o predošlej interakcii klieta s iným serverom. V prípade internetového obchodu môže byť aktuálnou reprezentáciou napr. stav nákupného košíka, ktorý obsahuje link na ďalší stav, objednávanie. Po kliknutí naňho dostane klient reprezentáciu, ktorá zodpovedá objednávke daného nákupného košíka, t.j. napr. vyplneniu adresy. Tá môže potom obsahovať link na potvrdenie, pri kt. človek dostane reprezentáciu zodpovedajúcu potvrdeniu objednania daného nákupného košíka na zadanú adresu, kde je link na zaslanie potvrdenia. Takto sa v princípe správa na stavovo uvedomelých reprezentáciách založená RESTovská komunikácia. Podstatné je, že každú požiadavku mohol vybavovať iný server, lebo neexistovala žiadna serverovská session. Tým pádom funguje korektne aj BACK button, ktorý pošle jednoducho request na predošlý krok transakcie zavolaním GET na príslušnej URL, ktoré ju reprezentovalo.

Praktický príklad Uvediem teraz krátky praktický príklad, kde bude principiálny prechod od slovieb(metód) k podstatným menám(zdrojom) manipulovateľným cez HTTP metódy a práca sa reprezentáciami zrejmejšia.

Predstavme si aplikáciu na adrese *www.shop.sk*, ktorá by mala definované nasledujúce webové metódy volateľné klientami:

```
GetAllItem()
GetItem()
DeleteItem()
AddItem()
```

V RESTovskej aplikácií by som mal prístup nie k funkciám, ale k zdrojom, a to napr. takto:

```
http://www.shop.sk/Item/[ID]
```

kde [ID] by predstavovalo konkrétne IDčko daného zdroja, buď napr. jeho názov, alebo databázové číslo, atď. Nad touto entitou by som mohol volať rôzne HTTP metódy, teda napr.

metoda:	URL:	v starom RPC ponimani:
GET	http://www.shop.sk/Item/15	GetItem15)
DELETE	http://www.shop.sk/Item/15	DeleteItem(15)
POST	http://www.shop.sk/Item/	CreateItem()
GET	http://www.shop.sk/Item/	GetAllItems()

Klient si môže takisto vyžiadať rôzne reprezentácie, napr.

```
GET /Item/Books/Cooking/15 HTTP/1.0
From: www.shop.sk
Accept: text/xml
```

kde odpoveď by bola napr.

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 2010 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h2>Rodinna kucharka<h2>
<p> ...popis...</p>
<a href="www.shop.sk/Item/Books/Cooking">Ine knihy z tejto kategorie:</a>
</body>
</html>
```

To iste by ale mohlo byť realizované aj ako XML formát, napr. requestom:

```
GET /Item/Books/Cooking/15 HTTP/1.0
From: www.shop.sk
Accept: html
```

Na ktorý by bola odpoveď:

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 2010 23:59:59 GMT
Content-Type: text/xml
Content-Length: 4848

<?xml version="1.0" encoding="ISO-8859-1"?>
  <Item type="book">
    <title>Rodinna kucharka</title>
    <description> ...popis...</description>
    <link href="www.shop.sk/Item/Books/Cooking">
     Ine knihy z tejto kategorie:
    </link>
  <Item>
```

REST tooling a podpora Pri uvedení akéhokoľvek štýlu, architektúry či technológie je logické pýtať sa, ako vyzerá tooling, t.j. podpora nástrojov uľahčujúcich vývoj pomocou daného prístupu. Kým pri SOAP-e je to stack konkrétnych komerčných riešení a špecifikácií popísaných komplexnými WS-*(WS-security, WS-reliability, WS-Discovery,...) pravidlami , pri RESTe je to čosi oveľa jednoduchšie - internetové štandardy. To znamená HTTP protokol, MIME formáty, formáty štruktúrovaného prenosu dát(XML, JSON) vyvinuté webové komponenty(cache, proxies, gateways,...), URL adresy, teda všetko veci, s ktorými vedia pracovať ako webové komponenty, tak aj užívatelia jednoducho. Vo svete frameworkov sa tiež začína diať pohyb smerom k podpore RESTu, medzi užitočné voľby patrí Ruby on Rails(Ruby), Restlet(Java), Symfony(Python), MS WCF Data Services(C sharp), Django, TurboGears RestController(Python). Pythonovským frameworkom sa budem venovať ďalej podrobnejšie.

3.4 Význam RESTu

O tom, aké výhody z hľadiska všeobecných pozitívnych softvérových vlasností REST prináša, už bolo písané. Tie by však existovali aj keby daná aplikácia stála sama o sebe mimo

webovej architektúry. S ňou má však tento prístup k distribuovaným aplikáciám hlbší zmysel. Po prvé, stávajú sa tak v pravom zmysle slova súčasťou webu. Kým na SOAPovských webmetódach založený prístup priniesol na web jednu URL, jeden endpoint, za ktorým sa skrývala funkcionalita, ku ktorej nemal špecifickej sémantiky neznalý klient prístup, pri RESTe sa každá entita nachádza skutočne na webe, pretože má svoje jednoznačné URL prístupné pre všetkých cez vopred dohodnuté CRUD rozhranie. Môže byť teda linkovaná z tej istej aj iných aplikácií, vďaka štandardnosti reprezentácií môže byž zobraziteľná obyčajným webovým prehliadačom, môže byť bookmarkovateľná a odovzdateľná tak ďalej. Vďaka pokročilej infraštruktúre môže aplikácia takto prirodzene ťažiť z jej možností ako cacheov pre zlepšenie interaktivity, firewallov pre bezpečnosť, a obyčajný webový prehliadač vie byť automaticky klientom k hociakej špecifickej webovej službe. Navyše využíva HTTP tak, ako bol vymyslený - ako aplikačný protokol, na rozdiel od SOAP-u, pre ktorý je HTTP len transferový protokol. Takisto podstatné je dodržiavanie sémantiky HTTP metód - teda na získanie zdroja sa používa správne GET a nie POST, tak ako je tomu pri SOAPe.

REST výborne zapadá do konceptu Webu 2.0. Moderná webová aplikácia(RIA - Rich web application) sa tak delí na AJAX-ový klientský engine schopný dynamicky obhospodarovať grafické užívateľské rozhranie a RESTovský backend schopný posielať jednak tento AJAXový kód a jednak reprezentácie entít, s ktorými potom pracuje. Podstatnou vlastnosťou webu 2.0 je veľká interaktivita a hlavne ľuďmi tvorený obsah. To je s jednoznačným rozhraním k zdrojom ďaleko jednoduchšie, pretože človek, príp. programátor zisťovať pri každej aplikácií zvlášť, ako s ňou pracovať. Prelinkovanosť vedie k ďalším možnostiam nachádzania prepojení entít, ktoré môžu referovať k tomu istému reálnemu objektu aj v rôznych systémoch a takisto (pomocou implementácie metódy GET) k dostupnosti pre webové crawlery a indexovanie nástroje, SEO, atď. Pretože na webe fakticky existuje to, čo je na webe nájditeľné, vyhľadateľné.

Ďalšou využitím RESTu je prirodzená integrácia webových aplikácií. Jedna aplikácia tak vďaka jednoduchosti rozhraní môže ľahko využívať služby inej, bez nutnosti nejakého konceptu objavovania služieb, tak ako je tomu u tradičných SOAPovských webservisoch. Môžu tak vznikať nové formy webových aplikácií poskytujúcich napr. zdroje zodpovedajúce zoskupeniu zdrojov z iných aplikácií, kde napr. k danej knihe nájdem jej ceny v kníhkupectvách, dostupnosť v knižniciach, komentáre na fórach, článok na wikipedii, link na stránku vydavateľstva, zoznam iných titulov toho istého autora, atď. Nič z tohto by možné nebolo, keby každý zo spomenutých informačných dodávateľov(obchody, knižnice, vydavateľstvo, fóra,...) implementoval vlastnú množinu rozdielnych metód.

4 PROGRAMOVACIE JAZYKY

Programovacie jazyky majú za sebou relatívne krátku históriu, počas ktorej ale stihli vzniknúť mnohé vetvy, paradigmy a prístupy. Uvediem teraz niekoľko možných pohľadov na delenie programovacích jazykov.

4.1 Historické delenie

Na začiatku bolo slovo. A to slovo bol assembler. Ten sa dá považovať(a pod „ním“ mám na myslim celé rodiny assemblerov) za prvý programovací jazyk. Vyniká rýchlosťou, lebo de facto sa píšu priamo inštrukcie pre procesor, ale je veľmi nevhodný na akékoľvek vyššie abstrakcie a časová náročnosť napísania relatívne jednoduchého programu je v ňom v porovnaní s modernými jazykmi obrovská. Patrí do druhej generácie programovacích jazykov. Prvú som schválne preskočil, lebo to boli prakticky priamo inštrukcie procesora a teda nie nejaký „jazyk“. Tretia generácia priniesla jazyky ako Cobol, C, C++, Basic, Java. Oproti jazykom druhej generácie pracovali na vyššej abstrakcii a znovapoužitelnosť kódu tu dostáva ovela väčší priestor. Štvrtá generácia sú doménovo závislé jazyky, určené na niečo konkrétne a teda sa už nedá vždy hovorit o programovacom jazyku v zmysle, že v ňom spraví človek čokoľvek. Typický príklad je dotazovací jazyk SQL. Piata generácia znamená už aj zmenu paradigmy a za tieto jazyky za označujú tie, ktoré nepracujú s algoritmami riešenia problému, ale s jeho popisom, deklarovaním. Toto delenie na generácie sa ale nie celkom zhoduje s historickým vývojom jazykov v nich zastúpených. Čoraz viac používané sú vysokoúrovňové jazyky ako C Sharp a Java, ktorě poskytujú veľmi rýchly vývoj s dostatočnou mierou abstrakcie a hlavne zásobárňou knižníc, takže rutinne opakované postupy si vyžadujú minimálny čas a programátorovi ostáva viac času na samotnú logiku aplikácie.

4.2 Delenie podľa spôsobu behu programu

Podľa spôsobu behu programu sa delia na kompilované a interpretované. Rozdiel je ten, že zatialčo po skompilovaní sa program stáva samostatným funkčným programom nad danou platformou, program písaný v interpretovanom jazyku potrebuje na svoju činnosť zvláštny program - interpreter. Ten sa potom stará o preklad na strojové inštrukcie priamo počas vykonávania programu. Takými jazykmi sú napr. jazyky Python, Ruby, Lisp. Kompilované sú napr. Cobol, alebo relatívne nízke jazyky C, C++, ktoré vymieňajú abstrakciu a pohodlný vývoj za rýchlosť. Špeciálnu kategóriu tvoria sčasti kompilované jazyky ako C sharp alebo Java, ktoré sa síce skompilujú do pseudokódu(bytode v prípade javy, CIL v prípad C sharpu), ale ten samotný sa vykonať nedá a na svoju činnosť potrebuje virtuálny stroj .NET, resp. JRE, ktorý sa potom musí vyvinúť pre každú konkrétnu platformu. Výhodou je prenositeľnosť kódu. Ďalšou často využívanou kategóriu sú skripovacie jazyky, ktorými sa dá ovládať činnosť nejakého systému, napr. PHP na kontrolu generovania stránok na webserveri, resp. Javascript v prehliadači.

4.3 Delenie podľa paradigmy

Paradigma je spôsob popisu programovacích problémov. Ako prvé sa uchytili procedurálne jazyky, používajúce k riešeniu konkrétne príkazy v podobe cyklov a práce s pamäťou. V 90-tych rokoch sa objavila paradigma objektovo orientovaného programovania, kde sa problém snaží modelovať cez logické objekty, ich vzťahy a interakcie. Veľkú skupinu tvoria deklaratívne jazyky. Tie sa fundamentálne odlišujú v tom, že namiesto AKO sa má niečo

spraviť, popíšu iba, ČO sa má vykonať, presnejšie, čo má platiť a je úlohou kompilátora, aby urobil všetky potrebné operácie. Sem patria funkcionálne jazyky a takisto logické. Existuje ešte kopec iných paradigiem, ako modulárna, paralelná, automatová, agentová, ale tie nie sú mainstream.

4.4 Delenie podľa účelu

Podľa účelu by sme mohli rozdeliť jazyky na 2 základné kategórie - problémovo špecifické a univerzálne. Tie prvé sú určené do istého prostredia, na riešenie určitej funkcionality a majú len obmedzený rozsah pôsobnosti, zato sú v danom prostredí veľmi efektívne. Príkladom je SQL, Matlab, PHP, atď. Univerzálne sú tie, v ktorých sa dá v princípe naprogramovať hocičo, ktoré majú teda dostatočné spojenie s nízkou úrovňou počítača.

4.5 Delenie podľa spôsobu typovania

Typovanie, teda definovanie typov, môže byť statické alebo dynamické. To prvé znamená, že typy sú určované už počas kompilácie a nemôže sa teda stať, že by sa s nim zaobchádzalo nevhodným spôsobom, napr. v snahe do čísla priradiť string, pretože takúto chybu zbadá už kompilátor. Naopak pri dynamicky typovaných jazykoch sa typ premennej určuje na základe hodnoty, ktorá je jej priradená.

5 PYTHON

Python je open-source, multiparadigmový, objektový aj funkcionálny, high-level, dynamický interpretovaný jazyk. Bol implementovaný Guido van Rossumom v Holandsku v r. 1989. Multiparadigmovosť bola jedna zo základných filozifických východisiek. Dynamickosť a interpretovanosť ho robia spolu s jednoduchosťou zápisu vhodným prostriedkom pre vývoj webových aplikácií, najmä preto, že výsledok zmeny programu si môže človek hneď overiť bez nutnosti čakať na kompiláciu, ako je to napr. u ASP. Vďaka silnej internetovej komunite za ním stojacou a vyvíjajúcej ďalšie utility a komponenty, je to vhodný nástroj ako pre prácu s webom, tak pre iné typy programovanie(databázové, GUI,...). Viac o ňom sa dá nájsť na [15].

5.1 Typické znaky pythonu

Medzi špecifiká pythonu patrí syntaktický význam úpravy zdrojového textu, kde odsadenia majú ten význam, ako v iných jazykoch zátvorky. Má automatickú správu pamäte, disponuje bohatou štandardnou knižnicou. Je rozšíriteľný pomocou C++ templatov, čo mu dodáva veľkú variabilitu. Obsluhovateľný vie byť ako volaním .py súborov, tak priamo z konzoly, kde sú príkazy okamžite interpretované a služi tak aj ako akási rozšírená "kalkulačka"pre okamžité rýchle operácie s dátami. Python chápe implicitne všetko ako objekt,

preto nie je problém pristupovať za behu napr. k typu danému objektu(názvu), pretože obsahuje referenciu na triedu, ktorá je takisto len objektom so svojím menom. Ako moderný jazyk podporuje exception handling a vďaka multiparadigmovosti aj lambda výrazy. Medzi natívne podporované typy patria všetky možné typy čísel(floaty, integery,...), stringy, a súbory. So stringami vie veľmi šikovne manipulovať(spájanie, vyhľadávanie, kopírovanie,...). Pre agregáciu poskytuje triedy list, teda typický zoznam, ktorého prvky ale nemusia mať rovnaký typ, potom dictionary, teda slovník, vhodný pre implemenáciu rôznych mapovaní a nakoniec tuple, ako jednoduché spojenie viacero samostatných hodnôt či objektov. Čo sa štandardnej knižnice týka, tá poskytuje funkcie pre prístup k súborom, matematickú funkcionalitu, podporu http protokolu, prácu s XML formátom a mnoho iných vecí, užitočných pri vývoji webu.

5.2 WSGI

Web Server Gateway Interface, definovaný na [4], predstavuje rozhranie medzi serverom a pythonovskou webovou aplikáciou. Server zavolá funkciu(ktorej názov si väčšina frameworkov, resp. WSGI-serverov určí napevno), ktorej predá 2 argumenty - objekt *environ*, obsahujúci všetky údaje o HTTP požiadavke, ktoré potrebuje aplikácia vedieť a callable(objekt implementujúci rozhranie callable, teda „zavolateľný“ objekt) *start_response*. Funkciou aplikácie, resp. frameworku je potom vytvoriť správu aj so statusom a hlavičkou, zavolať funkciu *start_response*, ktorej ako argumenty dá tento status a hlavičku a nakoniec vrátiť iterable objekt pomocou príkazu return. Príklad takejto jednoduchej aplikácie, ktorá vracia len „hello world“, je:

```
def moja_aplikacia(environ, start_response):
    status = '200 OK'
    response_headers = [('Content-type','text/plain')]
    start_response(status, response_headers)
    return ['Hello world\n']
```

5.3 Pythonovské frameworky

Hoci sa dá aplikácia vyvíjať priamo v pythone, je efektívnejšie budovať ju použitím frameworku, pretože ten vie mnoho typických vecí spoločných pre všetky weby zautomatizovať a zaobaliť. Všeobecná funkcionalita frameworkov už bola spomenutá vyššie. V tomto odstavci sa zmienim o konkrétnej funkcionalite vybraných pythonovských frameworkom a o tom, ako môžu byť použité pre tvorbu REST aplikácií. Kompletný zoznam je možné nájsť na http://wiki.python.org/moin/WebFrameworks. Tento obrázok(získaný z [2]) zachytáva poskytovanú funkcionalitu o najpoužívanejších frameworkov:

Project	Language	Ajax	MVC framework	MVC Push/Pull	i18n & l10n?	ORM	Testing framework(s)	DB migration framework(s)	Security Framework(s)	Template Framework(s)	Caching Framework(s)	Form Validation Framework(s)	Python 3.*
CherryPy	Python	No	controller & URL dispatching		Yes	ORM agnostic	use stdlib's unittest and doctest	depends on ORM		Templating engine agnostic	Yes	Form validation engine agnostic	Yes
Django	Python	Django uses jquery in the admin, but is js-agnostic in the user templates	Full Stack	Push	Yes	Django ORM	Yes	reusable applications which might get merged into core e.g. South	ACL-based	Django Template Language	Cache Framework	Django Forms API	No
Grok	Python	Yes	Yes	Pull	Yes	OODBMS called ZODB, SQLAlchemy, Storm	Unit Tests, Functional Tests	ZODB Generations	Yes	Yes	Yes	Yes	
Pyjamas	Python, Javascript	Yes	Use PureMVC python version (compiled to javascript)		Yes	???, no direct data access		No					No
Pylons	Python	helpers for Prototype and script.aculo.us	controller	Push	Yes	ORM-agnostic	via nose	depends on ORM		pluggable (mako, genshi, myghty, kid, etc.)	Beaker cache (memory, memcached, file, databases)	preferred formencode	No
TurboGears	Python	Toolkit-independent, provides support via JSON	Full stack, best-of-breed based	Push	Yes	SQLAlchemy	nose	No	Repoze.what & Repoze.who	Genshi, additional plugins available	Support for memcached, and any WSGI compliant system	ToscaWidgets, utilizing FormEncode	No
web2py	Python	Yes	Yes	Push	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Webware	Python	No	Optional	Pull	No	Yes	Yes	No	Yes	Yes	No	No	No
Zope 3	Python	via add-on products, e.g. Plone u/KSS	Yes	Pull	Yes	native OODBMS called ZODB, SQLObject, SQLAlchemy	Unit Tests, Functional Tests	ZODB generations	ACL-based	Yes	Yes	Yes	No

Frameworky môžu byť buď základné, poskytujúce mapovanie HTTP requestov na handlery, alebo tzv. full-stack, pokrývajúce komplexnú funkcionalitu. Medzi prvé patrí napr. CherryPy, medzi druhé django, alebo turbogears.

CherryPy

CherryPy je základný HTTP framework(dokumentácia na [12]). Umožnuje niekoľko variánt mapovania URL, okrem toho podporuje caching a správu sessionov. Základný URL handling vyzerá tak, že sa skonštruuje objektový strom, ktorý zodpovedá URL hierarchii, pričom to, čo sa nedá namapovať na názvy objektov a ich metód, sa namapuje na parametre metódy. Nasledovný kód to demonštruje:

```
class Root:
    def index(self):
        return "Hello!"
    index.exposed = True

class Branch:
    def index(self):
        return "Howdy"
    index.exposed = True

    def default(self, attr='abc'):
        return attr.upper()
    default.exposed = True

    def leaf(self, size):
        return str(int(size) + 3)
    leaf.exposed = True

root = Root()
root.branch = Branch()
app = cherrypy.tree.mount(root, script_name='/')
```

posledné riadky hovoria, že mapovanie začína tým, že URL "/"bude spracované triedou Root(metóda mount). Každá ďalšia URL bude mapovaná podľa zhodny názvov vnorených objektov a ich metód. Teda napr. "/branch"v objektovom strome prislúcha "root.branch", o ktorú sa stará trieda Branch. Ďalej "/branch/leaf"by sa namapovala na metódu leaf triedy Branch a "/branch/leaf/10"by dopadla takisto, ale "10"by sa odovzdalo tejto metóde ako parameter. Špeciálny význam majú funkcie ïndexä "default". Index je určená pre handling tých URL, ktoré končia na "/", teda napr. "/branch/". Ďalej metóda default je určená pre handling inak nedefinovaných URLiek, teda napr. "/branch/neexistujem". Premenná exposed naznačuje, že ma daná metóda byť prístupná ako URL handler, teda "viditeľná"na webe.

CherryPy a REST

Tento základný model URL handlingu však nie je až tak vhodný pre REST. CherryPy však poskytuje istú variabilitu pri volení metódy, akou sa budú mapovať URL. MethodDispatcher je to, čo sa v tomto prípade hodí viac. Ten dokáže mapovať nielen podľa objektového

stromu, ale aj priamo HTTP metódy. V takom prípade sa exposed zadáva pre danú triedu, nie metódu, ako naznačuje tento príklad:

```
class Root:
    exposed = True

    def GET(self):
        return repr(self.things)

    def POST(self, thing):
        self.things.append(thing)

root = Root()

d = cherrypy.dispatch.MethodDispatcher()
conf = {'/': {'request.dispatch': d}}
cherrypy.tree.mount(root, "/", conf)
```

Django

Django(dokumentácia na [13]) poskytuje štandardnú frameworkovú funkcionalitu, teda templating, modelovanie entít, caching, url dispatching, autentikáciu, validáciu, atď. Aplikácia je rozdelená na tzv. modely a viewy. Model je ako obvykle databázová entita, zatiaľčo view sa stará o zhotovenie odpovede, preberá teda funkciu kontrollera v MVC vzore. Ten sa v prípade djanga stáva MTV vzorom - model, template, view, kde template je zodpovedný za spôsob zobrazenia(reprezentáciu), zatiaľčo view za výber zobrazovanej entity. Modely sa definujú nasledovne:

```
class Trip(models.Model):
    title = models.CharField(max_length=200)
    desc = models.CharField
    id = models.IntegerField()
```

URL mapuje pomocou regulárnych výrazov. Tzv. grupy v rámci regulárnych výrazov predáva potom ako parametre metódam viewov.

```
urlpatterns = patterns('',
    (r'^trips/$', 'mysite.trips.views.collection'),
    (r'^trips/(?P<trip_id>\d+)/$', 'mysite.trips.views.detail'),
```

Toto mapovanie sa postará o to, že metóda index triedy Trips bude obsluhovať URL "/tripsä konkrétny trip na adrese "/trips/10"bude zobrazený pomocou metódy detail. Kód:

```
def collection(request):
    trips = Trip.objects.all()
    return render_to_response('trips/list.html', {'trips': trips})

def detail(request, trip_id):
    trip = Trips.objects.get(id=trip_id)
    return render_to_response('trips/detail.html', {'trip': trip})
```

V tejto ukážke je možné vidieť viacero vecí. Jednak je to jednoduchá práca s databázou, kde získanie kolekcie všetkých entít daného typu sa realizuje cez volanie "Nazov.objects.all()", a potom funkcie

`render_to_repsonse`

sa stará o automatické vygenerovanie a returnovanie zvoleného HTML templatu, pričom ako parameter berie dictionary hodnôt, ktoré sa majú v template prepísať. Templatovací jazyk pre djanko umožňuje takisto okrem priameho vkladania aj iteráciu a podmienené vkladanie.

Django a REST:

Výhodou djanga je, že obslužné metódy dostávajú na vstupe objekt HttpRequest, ktorý obsahuje informácie potrebné pre RESTovskú aplikáciu, teda napr. metódu alebo požadovaný typ reprezentácie. Podobne môže nastaviť HTTP headre v objekte HttpResponse, ktorý vracia. Vyššie uvedenú metódu by sme teda mohli upraviť napr. nasledovne:

```
def collection(request):
  if(request.method == "POST"):
    trip = Trip()
    trip.title = request.POST['title']
    trip.desc =  request.POST['desc']
    # vrat reprezentaciu tohto tripu
  elif if(request.method == "GET")::
  trips = Trip.objects.all()
  if request.meta[Content-Type]== "text/html":
   return render_to_response('trips/list.html', {'trips': trips})
  elif request.meta[Content-Type]== "xml":
    response= HttpResponse()
    response.write(createXml(trips))
    response.['Content-Type'] = 'text/xml:
    return response
  else:
   return HttpResponse("unknown representation")
```

takže by sa správala v závislosťi od toho, akú metódu by sme na kolekciu použili - GET alebo POST - a v prípade GETu by rozlišovala, aký typ reprezentácia si užívateľ žiadal.

Turbogears

Je framework vybudovaný podľa MVC vzoru(dokumentácia na [16]). Turbogears pozostáva z viacerých nezávislých súčastí. Základom je WSGI kontroller Pylons, ktorý sa stará o obsluhu požiadaviek a spracovanie URL requestov. Genshi je templatovací systém. SQLAlchemy je nástroj, vďaka ktorému je možné mapovať objekty na databázové tabuľky.

Vývoj aplikácie v TurboGears

Projekt turbo gears rozdeľuje aplikáciu na viacero modulov podľa časti funkcionality, ktorá sa ich týka(templaty, controllery, modely, security,...). Základom je root.py, kde sa definuje základné mapovanie všetkých URLiek. Koncept spracovania URL je nasledovný: najprv sa určí, ktorý kontroller je za danú URL zodpovedný. To sa deje tak, že sa matchuje objektový strom vychádzajúci z RootControlleru so žiadaným URL. Ak RootContoller obsahuje objekt trip = TripContoller(), kde TripContoller je trieda dediaca od Controllera, tak potom "(host)/trip"sa bude mapovať na metódy tohto TripControllera. Konkrétna metóda sa vyberie na základe matchovania jej názvu so zbytok URL, teda metóda ïntro"bude obsluhovať URL "(host)/trip/intro". Zvyšok URL sa predá ako parametre metódy. Keď teda zadefinujeme:

```
class TripController(Controller):
  @expose('myapp.templates.intro')
  def intro(self, title, text):
    return dict(pozdrav="ahoj")
```

Tak pri URL "(host)/trip/intro/new?text=blabla"sa zavolá táto metóda, pričom sa jej predajú parametre title="newä text="blabla". Atribút expose slúži na to, aby túto metódu sprístupnil ako webovú. Metóda môže vracať buď priamo HTML kód, alebo vygenerovaný HTML kód na základe templatu, ako v tomto prípade. Genshi je šikovný templatovací jazyk, ktorý v runtime vyhodnocuje pythonovské objekty, ktoré dostane od controllera a umožňuje napr. iteráciu, podmienené vkladanie a iné riadiace konštrukty.

Modely sa definujú takisto veľmi jednoducho:

```
class SampleModel(DeclarativeBase):
    __tablename__ = 'sample_model'

    id = Column(Integer, primary_key=True)
    data = Column(Unicode(255), nullable=False)
```

stačí ich oddediť od DeclarativeBase triedy, ktorú poskytuje SQLAlchemy. Tento model je pomocou SQLAlchemy uložený do tabuľky tak, že riadky predstavuju inštancie a stĺpce atribúty. SQLAlchemy podporuje hromadné ukladanie dát kvôli efektivite, takže zmenené dáta sú uložené až po zavolaní funkcie commit(). V rámci tohto prístupu sa dajú realizovať aj transakcie, teda zmeny vyžadujúce konzistentnú zmenu viacerých tabuliek naraz. TG podporuje viaceré bežné relačné databázy, priamo obsahuje SQLite, pričom má programátor prístup k dátam ako k objektom a nemusí sa namáhať s transormáciou do tabuliek alebo prepájaním databázy a aplikácie. TG umožňuje takisto validáciu pomocou atribútov zavesených priamo nad metódami kontrollerov a cez framework repoze aj autentifikáciu a autorizáciu.

Turbogears a REST:

Špeciálne pre vývoj RESTovských webových aplikácií prichádza TG s triedou RESTController. Tá je špecifická v tom, že na rozdiel od tried Controller alebo BaseController mapuje požiadavky nie na metódy, ktoré majú rovnaké meno ako časť URL, ale na metódy POST, GET, PUT a DELETE, ktoré sa volajú podľa HTTP metódy použitej na dané URL. Teda

napr.

```
class TripController(RestController):
@validate({'title':NotEmpty,
           'description':NotEmpty,}, error_handler=edit)
@expose()
def put(self, tripid, title, description):
    trip  = DBSession.query(Trip).get(trip_id)
    #update this trip.
```

sa zavolá, ak na URL "/trip/4884"zavoláme HTTP metódu PUT. ID 4884 sa predá funkcii ako parameter tripid, telo PUTu ako ostatné parametre. Pre ešte pohodlnejšie narábanie ponúka TG priamo funkcie

```
get_all()
```

a

```
get_one()
```

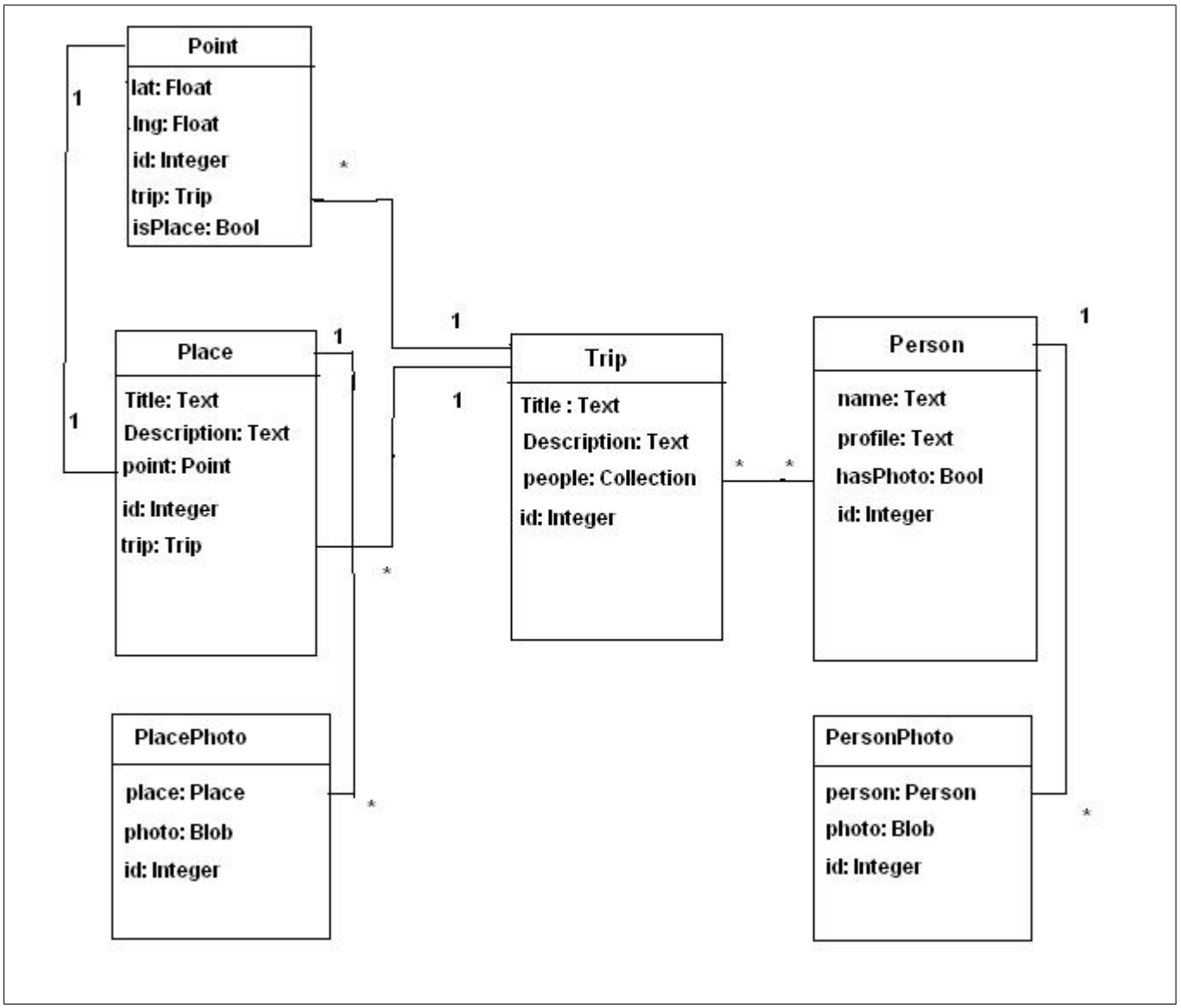
, ktoré implementujú metódu GET, pričom jedna sa volá pre URL na úrovni kolekcie(/trip) a druhá pre URL na úrovni konkrétneho resourcu(/trip/4848). Čo sa prístupu k HTTP requestom a responsom týka(kvôli HTTP headerom používaných v RESTe), Pylons sprostredkováva dva globálne objekty, request a response, ktoré zabaľujú Environ dictionary WSGI middlewaru. Takto môže programátor pristupovať priamo k HTTP headrom, ktoré potrebuje:

```
class TripController(RestController):
@expose()
def post(self, trip_id, title, description):
    #save this trip.
    #nastav HTTP status na created
    response.status = 301
    return ("/trip/"+trip.id)
```

6 Implementácia RESTovskej webovej aplikácie

Súčasťou tejto bakalárskej práce je implementácia jednoduchej webovej aplikácie navrhnutej podľa štýlu REST v jazyku Python. Ako námet som si zvolil výletný portál, ktorý by zhromažďoval informácie o rôznych výletoch rôznych ľudí. Podstatné bolo, aby sa obsah dal dynamicky vytvárať užívateľmi. Na technické detaily ako prihlasovanie, správa účtov a pod. som sa pri implementácií nesústredil, pretože témou práce nie sú pokročilé webové aplikácie ako také, ale štýl REST, šlo mi teda o implementáciu jeho kľúčových vlastností. V tejto aplikácií vystupuje niekoľko entít - budem používať aplickú terminológiu, ktorá sa v aplikácií použila - základom je trip(výlet). Jeho základnými atribútmi sú názov a popis. Pre trip si zvolí užívateľ na mapke body, ktoré reprezentujú trasu tohto tripu. Ďalej trip pozostáva z placeov, t.j. miest, pre ktoré je vytvorený samostatný záznam. Jeden place tak patrí práve jednému tripu, trip môže mať viacero placeov. K placeu sa okrem názvu a popisu dajú priradiť aj fotografie, ktoré môže užívateľ uploadovať. Ďalšou podstatnou entitou je osoba, teda person. Tá má meno, profil, a môže mať fotky. Vzťah medzi osobami a tripom je taký, že daného tripu sa môže zúčastniť viacero osôb a naopak daná osoba môže byť prítomná na viacerých tripoch. Aplikácia umožňuje pridávať tripy, pre daný trip zadať názov, popis, výber bodov na mapke a zúčastnené osoby. Pre vytvorený trip je ďalej možné pridať nové miesto, place, ku ktorému môže človek zadať názov a popis. Pre vytvorený place sa dajú ďalej pridať fotky. Ďalej je možné pridať osobu pomocou mena a profilu, a pre vytvorenú osobu uploadovať fotky. Informácie o každej z týchto entít(trip, place, person, fotky) je možné zobraziť, upravovať(nie nutne všetky atribúty) a mazať.

Na obrázku nižšie je vidieť UML diagram tried tejto aplikácie vyjadrujúci vzťah medzi spomínanými entitami.



Aplikácia bola vyvinutá pomocou webapp frameworku spoločnosti google a publikovaná na jej GoogleAppEngine infraštruktúre. Dôvody, prečo som si vybral práve toto riešenie, sú nasledovné:

- Šikovné mapovanie URL. Keďže REST je orientovaný na zdroje a tie sú adresovateľné cez URL, je potrebné, aby framework umožňoval flexibilné mapovanie URL na jednotlivé handlery. Webapp k tomu používa syntax regulárnych výrazov, takže sa vie vývojár ľahko pomocou grupovania(v regexoch) dostať k jednotlivým častiam URL, ktoré dostane ako parametre obslužnej metódy. Nie je tak problém robiť kaskádovité hierarchie URL, kde jedna resource je logická súčasť inej a jej URL teda môže obsahovať ako podčasť URL "vyššejëntity.
- Orientácia na HTTP. Webapp nekladie ďalšie vrstvy komponentovej abstrakcie nad HTTP a tak má vývojár plnú kontrolu nad priebehom komunikácie. Zabalením requestu

aj responsu do objektov umožňuje získavanie a nastavovanie HTTP headerov dôležitých pre REST. Navyše podporuje REST priamo v tom, že na metódy nemapuje len podľa URL, ale aj podľa HTTP metódy, čo značne uľahčuje implementáciu, pretože toto netreba zakaždým kontrolovať, stačí pre jednotlivé handlovacie triedy implementovať tie HTTP metódy, ktoré na danej URL chceme "prístupniť".
- Datastore. Keďže REST dobre slúži najmä na tvorbu bohatých aplikácií, kde majú užívatelia možnosť vytvárať obsah, je potrebné mať vhodné rozhranie na prácu s databázou, ktorá bude tieto vstupy ukladať. Na to výborne sluši Datastore, ktorý zabezpečuje ako objektový, tak aj relačný pohľad na dáta.
- Templatovací systém. Užívateľmi vytvorené dáta vytvárajú zdroje, ktorých reprezentácie môžu iní užívatelia požadovať. Preto je potrebné mať efektívne templatovanie, ktoré bude vedieť dáta získané z databázy(prípadne nejak transformované) vložiť do základných HTML reprezentácií.
- Google infraštuktúra. Ako už bolo zmienené, REST podporuje škálovatelnosť z hľadiska štýlu architektúry. Aby z toho mohla aplikácia ťažit, musí byť zasadená do prostredia schopného škálovatelnosti a to práve AppEngine poskytuje. Aplikácia môže narastať do veľkých rozmerov a o automatické pridávanie výpočtovej sily sa stará automaticky prostredie. Takisto je z hľadiska webdevelopmentu užitočné, že odpadá potreba konfigurácie HTTP serverov, pretože toto je zabezpečené automaticky. Aplikáciu stačí uploadnuť a to, čo ma človek možnosť skúšať lokálne, vidí v okamihu na webe. Komfortná je aj jednoduchá administrácia(schopná napríklad manuálne zasahovať do databázy), ako aj možnosť na login využívateľ google kontá, ak by to aplikácia vyžadovala.
Aplikácia sa funkčne delí na serverovskú a klientu časť.

6.1 Serverová časť

Server má na starosti spracovanie klientskych requestov. Je navrhnutý podľa vzoru Model-View-Controller:

Controller Ako kontroller slúži framework webapp. Ten sa v mnohom podobá djangovskej funkcionalite, a preberá s ním aj výhody blízkosti k HTTP protokolu, umožňujúce použiť ho na vytvorenie RESTovskej aplikácie. Funkcionalita webappu bola použitá na mapovanie jednotlivých URL na teda triedy zodpovedné za spracovanie žiadosti a vytvorenie odpovede. K tomu slúži webappom zabezpečený pohľad na request a response ako na objekty, nad ktorými sa dá pracovať(dostať napr. hodnoty odovzdané cez post, query string, atď.). Vytvorenie celej HTTP odpovede s príslušnými metadatami(headere) je už potom automatické. Priamo kontroller ale môže jednotlivé headre meniť, čo sa využilo pri nastavovaní formátu vrátenej reprezentácie. Webapp implementuje WSGI rozhranie, pričom ako parameter konštruktora inštancie WSGI aplikácie sa dáva práve mapovanie URL na jednotlivé triedy. Každá trieda musí implementovať HTTP metódy, ktoré očakáva, že môžu byť na danej URL volané. Tie sú potom volané automaticky podľa toho, aká HTTP metóda bola použitá.

Takto funguje mapovanie(neúplný úryvok kódu):

```
application = webapp.WSGIApplication(
         [('/trip/', TripHandler),
          ('/trip/(.*)/places/', PlaceHandler),
          (/trip/(.*)/places/(.*)/photo', PhotoHandler),
          ('/trip/(.*)/points', PointsHandler),
          ('/people/(.*)', PeopleHandler),
          ('/people', PeopleHandler),
          ],
          debug=True)
```

Pričom text zodpovedajúci jednotlivým zátvorkám (.*) sa potom predáva metódam ako parameter, takže napr. PlaceHandler zavolaný pre "/trip/48/places/"potom vie, že prvý parameter je 48(čo v tomto prípade označuje ID tripu). Takto:

```
class PlaceHandler(webapp.RequestHandler):
def post(self, * args):
...kod...
```

potom vyzerá implementácia pre HTTP metódu POST triedy, kt. sa stará o handlovanie požiadaviek pre place-y. To isté platí pre ostatné HTTP metódy a práve týmto prístupom je webapp vhodný pre REST.

Model Ukladanie jednotlivých entít umožnil Google DataStore, ktorý poskytuje rozhranie pre prácu s databázou. Každá trieda predstavujúca danú entitu bola oddedená od triedy db.Model. Tým získala metódy zaručujúce jej ukladanie do databázy, vyhľadávanie na základe ID a pod. DataStore takisto poskytuje SQL-like funkcionalitu cez vlastný dopytovací jazyk GoogleQueryLanguage, nad ktorým sa dajú tvoriť dotazy vyťahujúce dáta z databázy podľa syntaxe podobnej SQL. Takto napr. vyzera deklarácia triedy Place:

```
class Place(db.Model):
title = db.StringProperty()
desc = db.TextProperty()
trip = db.ReferenceProperty(Trip)
id = db.IntegerProperty()
point = db.ReferenceProperty(Point)
```

A takto:

```
person = db.GqlQuery("SELECT * FROM Person WHERE id="+str(id)).get()
```

môže vyzerať vytiahnutie objektu osoby podľa jej IDčka z databázy.

View Na generovanie html stránok odosielaných klientovi slúžil templatovací systém, pomocou ktorého bolo možné jednoducho vkladať jednotlivé vlastnosti objektov nachádzajúcich sa v kóde priamo do HTML kódu na vybrané miesta. Tento systém takisto poskytuje koncept dedenia, takže spoločný dizajn stránok umožňujúci navigáciu bol týmto ľahko vytvorený. Použitie je celkom priamočiare:

```
tripdetail.html:

<h4>Places:</h4>
{% for place in places %}
<p> <b>{{ place.title }} </b> -
<a href ="/trip/{{place.trip.id}}/places/{{place.id}}" >
place details
</a>
</p>
{% endfor %}


main.py:

pars = {"trip":trip, "places":places, "people":people}
self.response.out.write(template.render("tripdetail.html",pars))
```

Toto napr. nastaví ako výstupný HTML súbor tripdetail.html, pričom doňho doplní pomocou template.render funkcie jednotlivé parametre podľa toho, aké dáta v čase behu programu obsahujú(v tejto ukážke ide konkrétne o doplnenie názvov a linkov na place-y daného tripu).

6.2 Klientska časť

Klientska časť je tvorená javascriptovým kódom. Ten sa stará o dynamickú interakciu na základe vstupu užívateľa. Na stránkach zobrazujúcich detail danej entity môže človek kliknúť napr. na updatovanie alebo zmazanie tejto entity, príp. na stránkach zobrazujúcich kolekcie sa môže užívateľovi otvoriť formulár na vytvorenie novej entity. Všetka takáto interakcia na stránke vyžadujúca informácie zo servera(napr. zoznam osôb v systéme pre vytvorenie nového tripu, dotiahnutie bodov na mapku pre daný trip, atď.) je realizovaná pomocou AJAX-u. Ten pošle adekvátny GET príkaz na server a spracuje odpoveď vo formáte JSON.

Takto sa napr. realizuje získanie zoznamu ľudí, ak človek na stránke s kolekciou tripov klikne na zobrazenie formulára na vytvorenie nového tripu:

```
// získanie zoznamu ľudí
function getpeople() {
  upload('',readypeople,'GET','/people?page='+peoplepage)
  }

// vytvorenie zoznamu linkov na pridávanie ľudí
function readypeople(http_request) {
var name,id;
if ((http_request.readyState==4) && (http_request.status==200)) {
  document.getElementById("people").innerHTML="";
  peoplelist = eval("(" + http_request.responseText + ")").people;
  for (i=0;i<peoplelist.length;i++) {
    var textChild = document.createElement('a');
    textChild.href="javascript:SavePerson("+peoplelist[i].id+",'"+peoplelist[i].name
    textChild.innerHTML = peoplelist[i].name;
    document.getElementById("people").appendChild(textChild);
    document.getElementById("people").appendChild(document.createElement("br"));
    }
  }
}

// AJAXové volanie serveru
function upload(request, readyhandler, method, url) {
    var http_request = false;
    if (window.XMLHttpRequest) {
        http_request = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        try {
          http_request = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (eror) {
          http_request = new ActiveXObject("Microsoft.XMLHTTP");
        }
    }
    http_request.onreadystatechange = function() { readyhandler(http_request); };
    http_request.open(method,url, true);
    http_request.setRequestHeader('Content-Type', 'text/xml');
    if (request!='') http_request.send(request);
else http_request.send();
   }
```

Po kliknutí sa zavolá funkcia *getpeople*. Tá nastaví parametre AJAXového volania(metódu, URL, funkciu volanú na spracovanie odpovede). Následne zavolá *upload*, ktorá vykoná samotné volanie a po obdržaní odpovede zavolá funkciu *readypeople*, ktorá JSON odpoveď pomocou funkcie eval premení priamo na javascriptový objekt, z ktorého potom vyťahuje potrebné informácie.

6.3 CRUD rozhranie

REST požaduje, ako už bolo spomínané, aby každá entita mala priradené jednoznačné URL a implementovala metódy GET, POST, PUT, DELETE(aspoň niektoré). Dôležitou úlohou architekta je teda vymyslieť vhodnú syntax URL. Tie by mali zodpovedať logickej štruktúre entít v rámci aplikácie. Pod */trip* by človek očakával kolekciu tripov. Konkrétny trip by mal byť potom nájditeľný pod */trip/<id>*, kde <id> je nejaký jednoznačný identifikátor. Keďže názov sa mi nezdal dostatočne jednoznačný, rozhodol som sa k jednotlivým entitám pristupovať pomocou ich ID v rámci DataStore, pretože to je jedinečné. Link na konkrétny trip by teda mohol byť */trip/481*. Tripy majú priradené svoje place-y a pointy a ľudí, im zodpovedajúce adresy(pre tieto kolekcie) sú potom */trip/481/places*, */trip/481/points*, */trip/481/people*. Metódy PUT a DELETE sa vzťahujú vždy na konkrétnu konečnú entitu. GET sa môže použiť ako na konečnú entitu, tak aj na kolekciu entít. A nakoniec ku best practices patrí, že POST na vytvárania entít sa vytvára na kolekciu, do ktorej by táto entita patrila. Teda POST na URL */trip/* vytvorí nový trip so zadanými hodnotami. Tabuľka všetkých prístupných URLiek, metód, ktoré implementujú a sémantiky týchto volaní je nasledovná:

URL	Metóda	Význam
/trip	GET	vráť kolekciu tripov
/trip/<id>	GET	vráť konkrétny trip
/trip/<id>/places	GET	vráť kolekciu placeov daného tripu
/trip/<id>/places/<id>	GET	vráť daný place
/trip/<id>/people	GET	vráť zoznam ľudí na danom tripe
/trip/<id>/points	GET	vráť zoznam pointov na danom tripe
/trip/<id>/places/<id>/photo	GET	vráť zoznam fotiek daného placeu
/people	GET	vráť celkový zoznam osôb
/people/<id>	GET	vráť danú osobu
/people/<id>/photo	GET	vráť zoznam fotiek osoby
/people/<id>	PUT	uprav danú osobu
/trip/<id>/places/<id>	PUT	uprav daný palce
/trip/<id>	PUT	uprav daný trip
/people/<id>	DELETE	zmaž danú osobu
/trip/<id>/places/<id>	DELETE	zmaž daný palce
/trip/<id>	DELETE	zmaž daný trip
/people/	POST	vytvor osobu
/trip/<id>/places/	POST	vytvor palce
/trip/	POST	vytvor trip
/trip/<id>/places/<id>/photo/	POST	ulož foto daného placeu
/people/<id>/photo/	POST	ulož foto danej osoby

6.4 Reprezentácie

Vyššie uvedená tabuľka nehovorí, aká konkrétna reprezentácia bude pre dané URL zvolená. Tú si môže klient vyžiadať nastavením HTTP headera requestu. Aplikácia umožňuje okrem kanonickéj HTML reprezentácie dostať aj XML reprezentáciu daného zdroja. Preto je možné používať aj iných, bohatších a graficky prepracovanejších klientov, ktorí sú schopní pracovať s XML súbormi.

Takto môže vyzerať kanonická HTML reprezentácia:

Vylet do Karpat

Trasa:

Popis:

Sobotu rano sme vyrazili od zeleznej studienky...

new place

Places:

Zelezna - place details

Kacin - place details

People:

miro - person details

A tomu zodpovedajúca XML reprezentácia:

```
<Trip>
  <_desc>Sobotu rano sme vyrazili od zeleznej studienky...</_desc>
  <_id>154</_id>
  <_title>Vylet do Karpat</_title>
  <people>
    <href>/people/95</href>
    <href>/people/139</href>
    <href>/people/141</href>
  </people>
  <href>/trip/154</href>
  <places>
    <href>/trip/154/places/159</href>
    <href>/trip/154/places/160</href>
  </places>
  <points>
    <href>/trip/154/points/155</href>
    <href>/trip/154/points/156</href>
    <href>/trip/154/points/157</href>
    <href>/trip/154/points/158</href>
  </points>
</Trip>
```

7 Záver

V tejto bakalárskej práci bol vysvetlený pojem REST ako architektonický štýl pre vývoj moderných, škálovateľných webových aplikácií. Bola diskutovaná miera jeho podpory u jednotlivých pythonovských framewokov. Súčasťou práce bola implementácia ukážkovej aplikácie v pythone, ktorá bola v tomto texte v krátkosti popísaná. REST sa za posledné roky dostáva čoraz viac do popredia a získava veľa fanúšikov, ktorí jeho výhody dokážu oceniť. Príkladom implementácie je napr. rozhodnutie spoločností ako amazon alebo google poskytovať svoju funkcionalitu na RESTovskom základe. Web ako platforma má pred sebou ešte dramatický vývoj. Prakticky akýkoľvek systém môže mať ambíciu stať sa súčasťou webu. Výhody prepojenia zdrojov a funkcionalít naprieč webom budú viesť k čoraz väčšej integrácií a interaktivite, ktorá bude čoraz viac meniť spôsob, akým spolu komunikujú a spolupracujú ľudia, organizácie, firmy a ich klienti, štát a občania, atď. Blesková výmena informácií, multimédií a tvorba webového obsahu užívateľmi sa už teraz stávajú štandardom. Pre tento vývoj poskytuje REST vhodnú architektonickú bázu, na ktorej môžu byť moderné webové aplikácie postavené.

Literatúra

[1] *dmpc.dbp.fmph.uniba.sk/ rybar/it-software/docs/Integracia-SOA-REST.pdf.*

[2] *http://en.wikipedia.org/wiki/Comparison*$_o f_w eb_a pplication_f rameworks$.

[3] *http://www.livinginternet.com/i/ii.htm.*

[4] *http://www.python.org/dev/peps/pep-0333/.*

[5] *http://www.service-architecture.com/web-services/articles/.*

[6] *http://www.w3.org/CGI/.*

[7] *http://www.w3.org/Protocols/rfc2616/rfc2616.html.*

[8] *http://www.w3.org/TR/DOM-Level-2-HTML/html.html.*

[9] *http://www.w3schools.com/ajax/.*

[10] *http://www.w3schools.com/soap/default.asp.*

[11] *http://www.w3schools.com/xml/.*

[12] *www.cheerypy.org.*

[13] *www.djangoproject.com.*

[14] *www.json.org.*

[15] *www.python.org.*

[16] *www.turbogears.com.*

[17] Fielding, Roy T.; Taylor, R. N. *Principled Design of the Modern Web Architecture.* ACM Transactions on Internet Technology, 2002-2005.

[18] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures, Doctoral dissertation, University of California, Irvine.* 2000.