Vyhledávání

Komplexní podpora dotazování tvoří jednu ze základních funkcionalit databází. Oproti tomu většina běžných RESTových služeb nenabízí moc silné prostředky pro dotazování a omezuje se pouze na triviální předdefinované dotazy, příp. fulltextové vyhledávání. Ovšem mají-li webové služby IS sloužit aplikacím jako přímý [1] zdroj dat, tak je komplexnější podpora vyhledávání prakticky nezbytná.

Předdefinované dotazy

Všechny zdroje s parametrem v URI jsou v podstatě předdefinované vyhledávací dotazy. Kupříkladu /units/18000 vyhledá *organizační jednotku* s kódem *18000*. Na pozadí dojde k vygenerování *SELECTu* nad tabulkou nákladových středisek, kde kód střediska je rovný 18000. To je poměrně triviální dotaz. Trochu složitější se skrývá například za /programmes/MI/courses, který vyhledá všechny *předměty* patřící pod *studijní program* s kódem *MI*. Zde se vygeneruje polospojení nad programy, spojovou tabulkou a předměty, kde program má kód rovný MI.

Omezení takovýchto dotazů jsou zjevná. Co když potřebujeme například vyhledat všechny předměty, které se vyučují v zimním semestru, zajišťuje je Katedra softwarového inženýrství FIT a jejich název obsahuje slovo „prog“? Tady už potřebujeme nějaký dotazovací jazyk, který nám umožní kombinovat podmínky.

RSQL

UPOZORNĚNÍ: Integrace RSQL ještě není úplně dokončená, takže pro některé zdroje a konkrétní atributy nemusí fungovat správně!

RESTful Service Query Language (RSQL) je dotazovací jazyk a knihovna, jež jsem vyvinul pro KOSapi, která umožňuje vyhledávat záznamy (Atom Entry) podle jejich strukturovaných elementů (atributů) v Atom Content. Všechny zdroje KOSapi jsou koncipované tak, že Atom elementy využívají pouze pro metadata a vlastní data z KOSu jsou obsažená v Atom Content ve strukturované podobě (závisí na *Content-Type*, výchozí je XML). RSQL dotazy se v KOSapi překládají na SQL dotazy do KOSu.

Inspiroval jsem se Feed Item Query Language (FIQL), což je IETF návrh dotazovacího jazyka určeného (pouze) pro vyhledávání záznamů podle „metadat“ v Atom Entry. Syntaxe FIQL je výhodná svým prvoplánovým určením pro zápis v URI, díky čemuž ji není potřeba zakódovávat. Na druhou stranu je tím poněkud neobvyklá a ne příliš intuitivní. Jelikož jsem si stejně musel napsat vlastní parser, rozhodl jsem se tuto syntaxi využít a rozšířit ji ještě o alternativní zápis.

Proč jsem vlastně vyvíjel vlastní řešení a nevyužil nějaké standardizované? Důvod je prostý, žádné takové kupodivu zatím neexistuje nebo jsem ho nenašel. Tedy kromě standardu Open Data Protocol, který mimo jiné zahrnuje komplexní podporu pro dotazování. Ovšem využití OData pro KOSapi jsem z několika důvodů zavrhl a vzhledem k tomu, že podpora vyhledávání je jeho „inherentní“ součástí, tak mi její *samostatné* využití nepřišlo přínosné.

Gramatika a sémantika

RSQL výraz se skládá z jednoho či více *kritérií*, které se spojují logickými (Booleovskými) operátory.

```
expression = [ "(" ],
             ( constraint | expression ),
             [ logical-operator, ( constraint | expression ) ],
             [ ")" ];
```

Logické operátory jsou:

- AND : ”;” podle FIQL, nebo alternativní ” and ”
- OR : ”,” podle FIQL, nebo alternativní ” or ”

```
logical-operator = ";" | " and " | "," | " or ";
```

Operátor AND má standardně přednost, tj. všechny operátory OR se vyhodnocují až po něm. Toto chování lze samozřejmě změnit pomocí uzávorkování výrazů.

Kritérium se skládá ze selektoru, který identifikuje element v Atom Content, operátoru porovnání a argumentu.

constraint = selector, comparison-operator, argument;

Operátory porovnání jsou:

Název	FIQL	Alternativní	Platné datové typy
rovná se	==	=	textový řetězec, číslo, datum, výčtový typ, XLink
nerovná se	!=	!=	textový řetězec, číslo, datum, výčtový typ, XLink
menší než	=lt=	<	číslo, datum
menší nebo rovno	=le=	<=	číslo, datum
větší než	=gt=	>	číslo, datum
větší nebo rovno	=ge=	>=	číslo, datum

comparison-operator = "==" | "=" | "!=" | "=lt=" | "<" | "=le=" | "<=" | "=gt=" | ">" | "=ge=" | ">=";

Selektor odpovídá názvu elementu v Atom Content nebo jeho relativní cestě, pakliže je zanořený. Může také obsahovat „dereferenci“ XLink vazby pomocí tečkové notace.

```
selector   = identifier, { ("/" | "."), identifier };
identifier = ? ["a"-"z","A"-"Z","_","0"-"9","-"]+ ?
```

Argumenty mohou být dvojího typu. Libovolná sekvence znaků uzavřená mezi jednoduché či dvojité uvozovky, nebo sekvence znaků bez mezer, kulatých závorek, čárek a středníků.

```
argument    = arg_ws | arg_sq | arg_dq;
argument-ws = ? ( ~["(", ")", ";", ",", " "] )+ ?;
argument-sq = ? "'" ~["'"]+ "'" ?;
argument-dq = ? "\"" ~["\""]+ "\"" ?;
```

Porovnávání textových řetězců nezohledňuje velikost písmen (je *case insensitive*). Pokud je URL parametr multilang nastaven na true, tak zohledňuje texty v obou jazycích (neplatí pro dereferencované atributy). V opačném případě vyhledává pouze ve zvoleném jazyce (podle Accept-Language, nebo lang).

Při porovnávání řetězců lze využít i *divoké karty* a hledat pomocí nich i jen podle části řetězce. Způsob zápisu je stejný jako v SQL *LIKE*, pouze s tím rozdílem, že místo % se zde používá *. Například podmínce name=prog_am* vyhoví všechny předměty, jejichž název začíná na „prog“, následuje jeden libovolný znak, pak „am“ a cokoli (opět bez ohledu na velikost písmen).

V případě elementů, které reprezentují výčtový typ, je nutné jako argument uvádět *výčtový název* (enum), nikoli jeho lokalizovaný popis.

Argumentem pro XLink je identifikátor záznamu použitý v URI, což většinou bývá kód, nebo ID.

„Dereference“ vazeb (aka JOIN)

V dotazu je možné přistupovat i k atributům *odkazovaných* zdrojů (na které vede XLink) pomocí tzv. „dereference“. Jinak řečeno umožňuje zápis podmínky s *implicitním* spojením (*JOINem*) entit, mezi kterými existuje explicitní průchozí vazba (obdobně jako v HQL). Vazbami se prochází pomocí tečkové notace a je možné i zanořování. Kupříkladu unit.unitType==FACULTY vybere všechny záznamy, které jsou ve vztahu s organizační jednotkou *typu* fakulta. Na pozadí dojde k vygenerování polospojení (*LEFT JOIN*) tabulky předmětů s tabulkou nákladových středisek a podmínky unitType=FACULTY.

Mějte prosím na paměti, že tyto dotazy mohou generovat velkou zátěž databáze. Jakmile bude KOSapi napojené přímo na KOS, bude tento problém dost citlivý. Používejte je proto obezřetně a vyhýbejte se zbytečně neefektivním dotazům. Z těchto důvodů jsem také omezil maximální počet implicitních *JOINů* pro dotaz na 3.

Parametry

RSQL výraz se zapisuje do URL parametru query a je možné ho efektivně kombinovat s parametry offset, limit a orderBy.

Příklady

- /courses?query=name==*prog* - vrátí předměty, jejichž název obsahuje „prog“
- /courses?query=name=='programování v*' - vrátí předměty, jejichž název začíná na „programování v“
- /courses?query=credits>5 - vrátí předměty za více než 5 kreditů
- /courses?query=season==WINTER;(completion==CLFD_CREDIT,completion==CREDIT) - vrátí předměty, které se vyučují v zimním semestru a jsou zakončené klasifikovaným zápočtem nebo zápočtem
- /courses?query=department.unitType==FACULTY - vrátí předměty, které zajišťuje přímo libovolná fakulta (tzn. organizační jednotka typu fakulta)
- /teachers?query=extern==true&orderBy=lastName&limit=50 - vrátí vyučující externisty, seřadí je podle příjmení a výstup omezí na max. 50 záznamů

Pár slov k implementaci

RSQL jsem vyvinul speciálně pro KOSapi, ale jeho návrh a implementace je dostatečně obecná i pro použití v jiných RESTových službách postavených nad relační databází. Skládá se ze dvou navazujících knihoven.

První je *RSQL-parser*, který provádí lexikální analýzu, parsování a sestavení objektové reprezentace zadaného RSQL výrazu. Součástí je gramatika zapsaná v JavaCC, ze které je vygenerován vlastní parser.

Druhou knihovnou je *RSQL-hibernate*. Ta zajišťuje převod dotazu na Hibernate Criteria Query (objektová reprezentace HQL, resp. SQL dotazu), z něhož se následně generuje SQL dotaz do relační databáze. V tomto procesu hrají hlavní roli *RSQLCriteriaBuilder*, sada *CriterionBuilders* a *Mapper*. *CriteriaBuilder* prochází strom výrazu, generuje *Criterion* pro logické výrazy (AND, OR) a deleguje *kritéria* (porovnání) na odpovídající *CriterionBuilder*. Ty má připravené v kolekci, jíž iteruje dokud nenalezne takový, který umí obsloužit daný selektor a operátor. Kromě obecného *CriterionBuilder* obsahuje například takový, který umí vytvořit *Criterion* pro atribut vazby (i s NaturalID), multijazyčný text, selektor s implicitním JOINem, příp. speciální pro nestandardní entity. *Mapper* zajišťuje mapování *selektorů* (názvů v XML, příp. cest) na názvy příslušných *atributů* v entitách. Většina odpovídá 1:1, ale v některých případech je nutné použít přemapování (např. multijazyčné texty).

Obě knihovny jsem uvolnil pod licencí LGPL a umístil na GitHub - RSQL-parser a RSQL-hibernate.

[1] To znamená, že aplikace si nebudou uchovávat lokální kopii celé ani části databáze IS (cache se tím nevylučuje), ale budou je přímo získávat z webové služby.