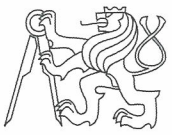
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Rozšíření produktu Semanta o zpracování metadat platformy IBM Cognos BI
Student:	Bc. Vladimír Kroupa
Vedoucí:	Mgr. et Bc. David Voňka
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

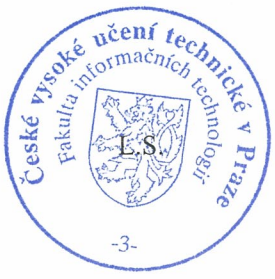
Pokyny pro vypracování

Semanta je aplikace podporující tvorbu znalostních bází. Cílem práce je analýza, návrh a implementace zobrazení metadat platformy IBM Cognos BI. Semanta je postavena na platformě XF3, která má schopnost vytvářet záznamy na základě metadat získaných pomocí SQL. Diplomová práce má tudíž dvě části: získání metadat z Cognos a jejich uložení do relační DB a vytvoření předpisu pro transformaci těchto dat na objekty v Semanta/XF3.

1. Popište platformu XF3, na ní postavenou aplikaci Semanta a její použití pro dokumentaci reportů.
2. Popište platformu IBM Cognos BI, zaměřte se zejména na reporty a jejich metadata.
3. Analyzujte možnosti získávání metadat o reportech z platformy Congnos. Popište strukturu metadat, která lze získat.
4. Navrhněte vhodný způsob jejich prezentace v aplikaci Semanta.
5. Implementujte modul, který bude řešit přenos metadat z platformy IBM Cognos do relační databáze.
6. Implementujte zobrazení těchto metadat v aplikaci Semanta.
7. Řešení otestujte a zdokumentujte.

Seznam odborné literatury

IBM Cognos Business Intelligence, ISBN : 1849683565, ISBN 13 : 9781849683562, Author(s) : Dustin Adkison



Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 13. října 2016

Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Rozšíření produktu Semanta o zpracování metadat platformy IBM Cognos BI

Bc. Vladimír Kroupa

Vedoucí práce: Mgr. et Bc. David Voňka

16. února 2017

Poděkování

Děkuji Davidu Voňkovi za cenné rady a za připomínky, které zásadně ovlivnily tento text. Dále děkuji kolektivu autorů PlantUML v čele s Arnaudem Roquesem za software, který výrazně usnadnil tvorbu diagramů pro tuto práci. Nakonec bych rád poděkoval komunitě okolo XML databáze eXist za ochotné zodpovídání dotazů.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na základě níž se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 16. února 2017 .....

Odkaz na tuto práci

Kroupa, Vladimír. *Rozšíření produktu Semanta o zpracování metadat platformy IBM Cognos BI.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Zásadní prvek v rozhodování současných firem představuje reporting. Reporting je vytváření informačních sestav nad daty, které má podnik k dispozici. Běžná velká firma má tisíce reportů, vytvořených z velké části v reportingové platformě, jako je IBM Cognos. Katalogizace a dokumentace reportů je nezbytnou součástí náplně práce reportingových oddělení.

V této práci mapujeme metadata IBM Cognos, pomocí loaderu implementovaného v Javě tato metadata zapisujeme do relační databáze a prezentujeme je ve webové aplikaci na katalogizační platformě XF3. XF3 zajišťuje vyhledávání a možnost manuálního doplňování vzniklé dokumentace. Pomocí Semanta Air zajišťujeme dostupnost vzniklé dokumentace přímo v platformě IBM Cognos.

Vzniklé řešení je předstupněm ke komerčnímu řešení, které bude nabízeno firmou Semanta.

Klíčová slova IBM Cognos Business Intelligence, Semanta XF3, business intelligence, integrace, XML

Abstract

Reporting, which involves querying data sources to produce a human readable report, is a crucial aspect of decision making in today's organizations. It is not unusual for a reporting department of an organization to operate with thousands of different reports, most of them created in a specialized reporting software. Cataloging and documenting of these reports are integral activities these departments carry out on a daily basis.

In this thesis we examine the metadata of the IBM Cognos platform. Using a metadata loader implemented in the Java programming language we insert the acquired metadata into a relational database. These data are then presented using web applications implemented on the specialized platform XF3. XF3 enables users to perform a full-text search upon the automatically created documentation and also gives users the ability to enrich this documentation manually.

The solution resulting from this thesis is a major stepping stone towards a commercial product that will be eventually offered by the Semanta corporation.

Keywords IBM Cognos Business Intelligence, Semanta XF3, business intelligence, integration, XML

Obsah

Seznam obrázků

Seznam fragmentů kódu

Úvod

Zásadní prvek v rozhodování současných firem představuje *reporting*. Reporting je vytváření informačních sestav (*reportů*) nad daty, které má podnik k dispozici. Reporty slouží k podpoře rozhodování managementu firmy. Typickým reportem může být přehled zisku z prodeje zboží podle kvartálů a podle zemí či regionů.

V podniku může v závislosti na jeho velikost a vyspělosti existovat stovky až tisíce reportů, zabývajících se rozličnými aspekty jeho fungování a sloužících k podpoře rozhodování v různých oblastech řízení. V jeden moment se tedy *business intelligence (BI)* oddělení velkých firem setkávají s nutností zabývat se problematikou katalogizace svých reportů. Mimo katalogizace, která spočívá primárně ve sběru metadat o reportech, se objevují nad velkou bází reportů další problémy, které je vhodné adresovat – například zadávání požadavků na modifikaci reportů.

Tato práce se zabývá katalogizací reportů vzniklých na platformě *IBM Cognos Business Intelligence* (dále jen IBM Cognos), která je podrobněji popsána v kapitole 1. Cílem je vytvořit ve vhodném software katalog reportů, který bude umožňovat snadné vyhledání jednotlivých reportů a bude přispívat k jejich pochopení. Kostra tohoto katalogu vznikne na základě dat získaných automaticky z platformy IBM Cognos. Na tuto kostru pak musí být možné v katalogu reportů přidávat další údaje. Mezi tyto údaje patří například vysvětlení struktury reportu a jeho účelu, kontakt na jeho autora, odkaz do slovníku použité terminologie (např. vysvětlení, co přesně chápeme pod pojmem „zisk z prodeje“) nebo screenshoty reportu.

Takto katalogizované informace o reportech by měly být přímo přístupné i z jednotlivých reportů v IBM Cognos.

Pro realizaci katalogu reportů jsme se rozhodli použít platformu XF3 od firmy Semanta. XF3 je webová platforma pro snadné vytváření formulářových aplikací, které mohou být vyplňovány jak automatickým nahráním z externího zdroje dat, tak manuálně uživatelem. V našem případě budou automaticky nahrávány základní údaje o reportu, získané z IBM Cognos a uživatelé budou manuálně vyplňovat doplňující informace zmíněné výše. Platforma XF3 obsahuje i komponentu Air, která nám umožní dopravit informace z katalogu přímo do reportu v Cognosu. Platformu XF3 blíže popisuje kapitola 2.

Implementační část práce se skládá ze dvou částí. První část práce budeme dále označovat jako *loader*. Loader provede získání dat z IBM Cognos přes aplikační rozhraní této platformy a po jejich zpracování provede jejich uložení do databáze. Ta je totiž v rámci platformy XF3 primárním zdrojem pro nahrávání externích dat. Loaderu se podrobněji věnuje kapitola 3, jejíž součástí je i návrh vhodných relačních struktur pro uložení dat loaderu.

Druhou část implementace označujeme jako *katalog reportů* nebo *metadata katalog*. Ta je realizována v rámci platformy XF3 a navazuje na loader tím, že zahajuje svůj životní cyklus načtením jím uložených dat z relační databáze. Této části se věnuje kapitola 4.

Plaforma IBM Cognos poskytuje aplikační rozhraní (dále jen API) pro jazyk Java a platformu .NET, v práci je použita verze pro jazyk Java. API je z pohledu jeho konzumenta poměrně nepřátelské. Vyznačuje se nízkou úrovní abstrakce – definuje pro požadavky a odpovědi API Javové objekty, které jsou ale v některých případech pouze obálky na XML dokumenty, které musí vytvářet a zpracovávat sám konzument API. API má dále tu vlastnost, že získávání dat z jednotlivých komponent platformy je nutné provádět odděleně. Pro získání kompletní sady informací o reportu je nutné provést volání tří různých komponent a odpovědi jednotlivých komponent ve formátu XML sjednotit do jednoho výsledku. Z tohoto důvod je značná část práce tvořena dotazováním se nad XML dokumenty.

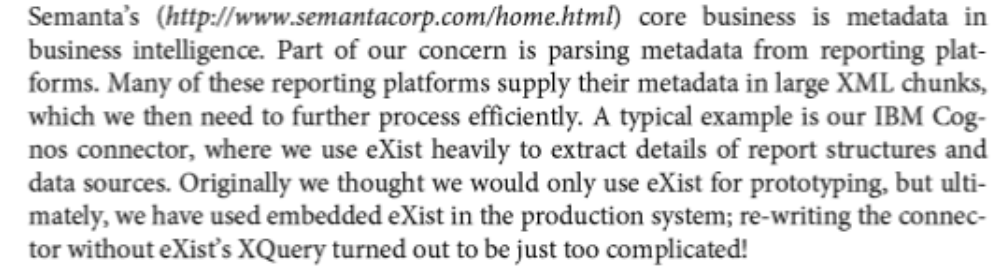
Pro tento účel je v práci využit jazyk XQuery, který umožňuje pokládat dotazy nad více XML dokumenty současně, což je nezbytné pro agregaci odpovědí jednotlivých komponent API. Pro provádění XQuery dotazů se v práci využívá nativní XML databáze eXist.

V první iteraci práce byla celá komponenta loaderu implementována jako XQuery aplikace pro eXist. Tato XQuery aplikace dostala na vstupu XML dokumenty získané z Cognos API a provedla jejich zpracování včetně závěrečného zápisu výsledků do relační databáze. Vyjadřovací možnosti jazyka XQuery toto umožňují, ale některé obraty (zejména týkající se použití relačních databází) jsou v XQuery poněkud těžkopádné. Z tohoto důvodu byla v druhé iteraci práce logika loaderu přepsána do jazyka Java a databáze eXist se používá pouze jako XQuery engine při komplikovanějších operacích nad XML dokumenty.

Komponenta katalog reportů je implementována na platformě XF3. Skládá se z *XF3 aplikací* a *nahrávacích úloh*, které definují mapování dat pro přenos ze zdrojové relační databáze do jednotlivých XF3 aplikací. Aplikace i nahrávací úlohy se definují v XML dokumentech. XF3 umožňuje do řady elementů v obou typech XML dokumentů vkládat výrazy v šablonovacím jazyce Velocity, které jsou dynamicky vyhodnocovány za běhu, což zvyšuje vyjadřovací sílu platformy. Každá XF3 aplikace operuje s jedním typem objektu a převážná část konfigurace konkrétní XF3 aplikace se zabývá právě definicí atributů tohoto objektu. Hlavní XF3 aplikací je v našem případě aplikace, která definuje objekt reportu. Objekt report má atributy jako jsou název reportu, URL reportu, popis reportu, seznam sekcí reportu, datum poslední změny reportu a další atributy.

Součástí platformy IBM Cognos je webová komponenta Report Viewer pro prohlížení reportů. Komponenta Air platformy XF3 umožňuje stránku reportu v Report Vieweru obohatit o informace z katalogu reportů (viz obrázek A.2). K tomu je zapotřebí splnit několik podmínek, které popíšeme v kapitole 4.

Díky využití XML databáze eXist v rámci této práce se autor dostal kontaktu s její komunitou. Zúčastnil se workshopu o eXistu v rámci konference XML Prague 2014. Na základě tohoto kontaktu vznikla zmínka (viz obrázek 0.1) o této práci v knize *eXist: A NoSQL Document Database and Application*


Semanta's (*http://www.semantacorp.com/home.html*) core business is metadata in business intelligence. Part of our concern is parsing metadata from reporting platforms. Many of these reporting platforms supply their metadata in large XML chunks, which we then need to further process efficiently. A typical example is our IBM Cognos connector, where we use eXist heavily to extract details of report structures and data sources. Originally we thought we would only use eXist for prototyping, but ultimately, we have used embedded eXist in the production system; re-writing the connector without eXist's XQuery turned out to be just too complicated!

Obrázek 0.1: Ukázka z knihy [1], zmiňující tuto práci

Platform [1].

Kapitola 1

Platforma IBM Cognos BI

Cílem práce je vytvořit katalog reportů, který umožní katalogizovat reporty podniku, které se nachází v BI platformě IBM Cognos. Tato aplikace bude sloužit primárně uživateli, který je konzumentem reportů (není tedy jejich autorem). Tato perspektiva nám usnadní rozhodování, které objekty BI platformy chceme katalogizovat a jaké informace nás o těchto objektech zajímají.

V textu budeme dále používat pro ilustraci demonstrační reporty, dodávané s platformou IBM Cognos. Právní důvody nám neumožňují využívat pro ilustraci reálné firemní reporty.

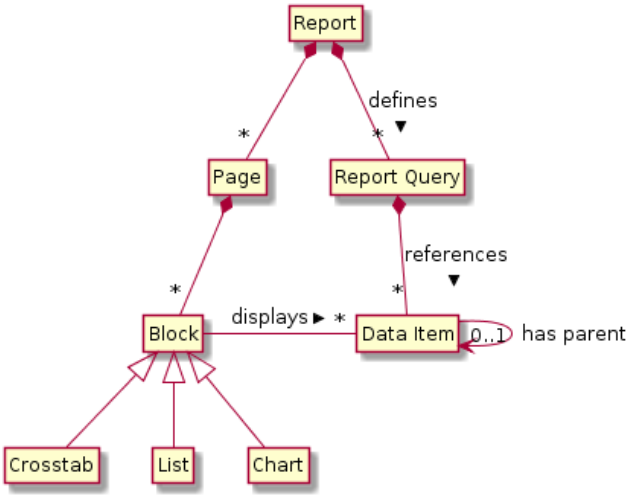
Platforma Cognos se skládá z řady aplikací. Konzumenti reportů využívají pouze aplikaci *Cognos Connection* pro prohlížení reportů. Dále jsou součástí platformy aplikace sloužící k vytváření obsahu, jejichž funkce se do značné míry překrývají. Aplikace pracují se stejnými objekty, jako jsou např. reporty a queries. Smyslem této duplikace je, aby méně techničtí BI uživatelé využívali jednodušší aplikace, jako je např. *Cognos Analysis Studio* a aplikace s nejvíce možnostmi používali autoři reportů a administrátoři. Pro potřeby zkoumání vlastnostmi objektů platformy nás zajímají „plnohodnotné“ aplikace *Cognos Report Studio* pro tvorbu reportů a *Cognos Framework Manager* pro vytváření a správu datových modelů.

1.1 Report

Z pohledu konzumenta BI platformy je primárním objektem zájmu report. Pod pojmem report rozumíme informační sestavu nad daty. Na obrázku A.1 vidíme report *2011 Sales Summary*, jak jej zobrazuje komponenta *Cognos Connection*. Tento report zachycuje objemy prodeje zboží za rok 2011 podle oblastí, typu zboží a zaměstnanců – prodejců. Je vizuálně rozdělen do tří základních částí, které nazýváme *sekce* reportu. V horní části je hlavička s názvem reportu. Dole je patička s datumem, číslem stránky reportu a časem, kdy byl report vygenerován. Uprostřed je tělo reportu, které obsahuje šest objektů – čtyři grafy a dvě tabulky. Tyto objekty souhrnně nazýváme (vzhledem k absenci univerzálně přijímaného termínu) jako *bloky* reportu. Kromě *grafu (chart)* a *tabulky (list)* existuje v IBM Cognos ještě třetí typ bloku – *kontingenční tabulka (crosstab)*.

Každý blok reportu obsahuje datové položky (*data items*). Při pohledu do pravé horní části reportu A.1 vidíme graf s názvem *Europe*. Na ose Y grafu jsou vyneseny dvě datové položky – *Gross profit* (hrubý zisk) a *Revenue* (příjem). Osa X představuje datovou položku *Quarter* (kvartál roku). Datové položka mají typy, vyjadřující jejich účel. V grafu *Europe* jsou hrubý zisk a příjem faktové datové položky, kvartál roku je položka typu dimenze.

Při editaci reportu v aplikaci *Report Studio* zjistíme, že datové položky jsou nadefinovány na úrovni dotazů (*queries*). Jednotlivé bloky reportu poté referencují datové položky z konkrétní query.



Obrázek 1.1: Struktura reportu

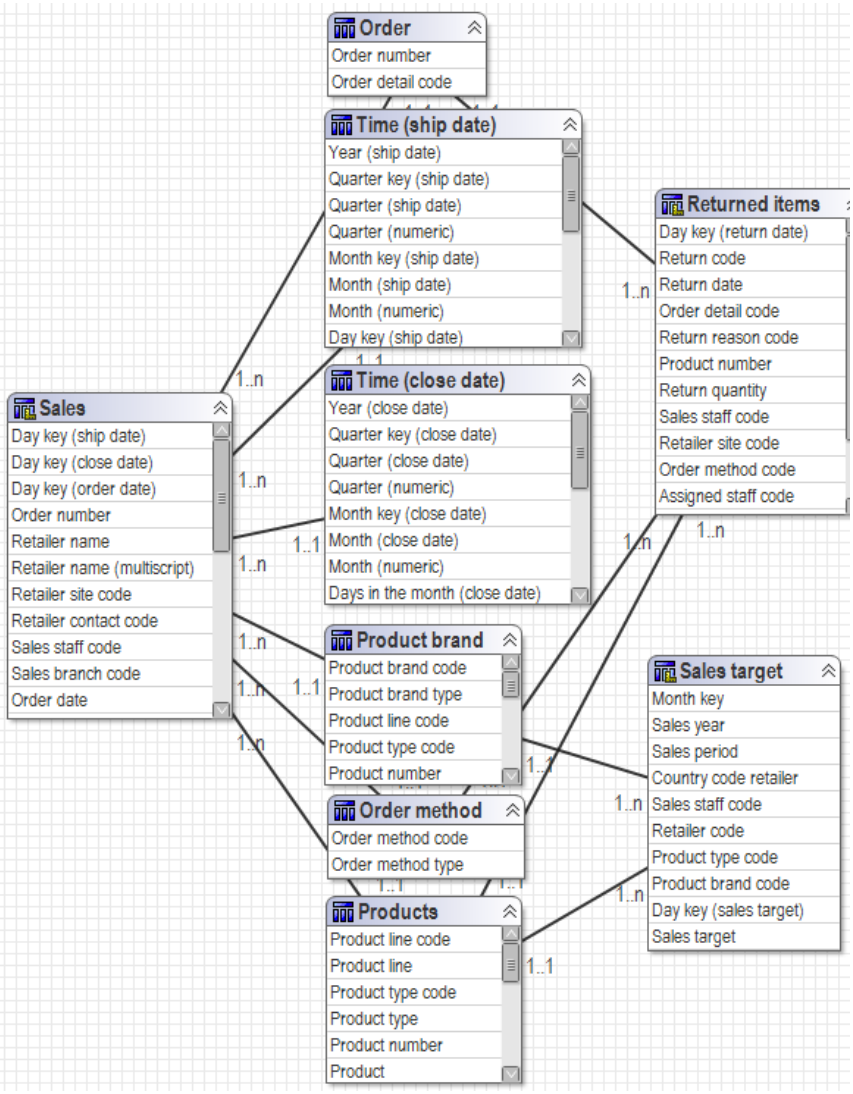
Obrázek 1.1 shrnuje výše uvedené poznatky o struktuře reportu. Reporty obsahují bloky. Bloky jsou seznamy, grafy a kontingenční tabulky. Bloky zobrazují datové položky. Datové položky jsou definovány v rámci queries.

1.2 Datový model (Framework Manager model)

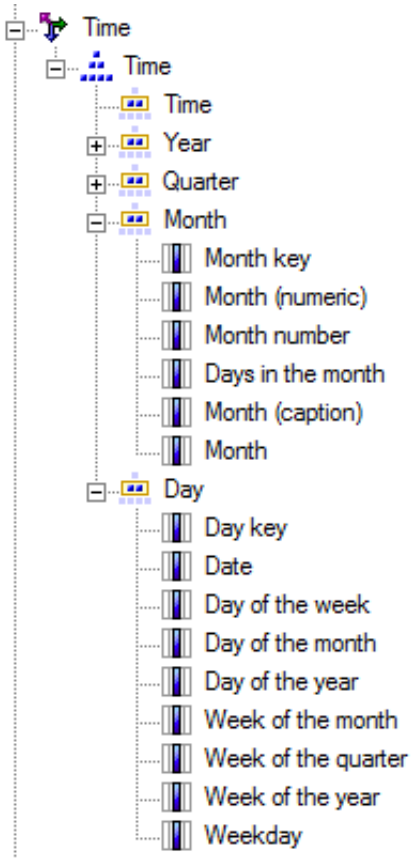
Datové položky reportu odkazují na *objekty datového modelu (package objects)*. Data, která report zobrazuje, tedy pocházejí z datového modelu (v terminologii IBM Cognos nazývaného *Framework Manager model*). Datový model je abstrakcí nad datovým zdrojem – například relační databází nebo OLAP kostkou. Na obrázku 1.2 vidíme diagram části datového modelu *go_sales*, jak jej zobrazuje aplikace *Framework Manager*. Objekty tohoto modelu používá výše popsaný report *2011 Sales Summary*.

Diagram na obrázku 1.2 zobrazuje entity – *query subjects* (odpovídající pojem v češtině by mohl být *výsledek dotazu*). Query subjects jsou logické celky, které odpovídají výsledkům dotazu. Query subject se skládá z *query items* (ekvivalent v češtině by mohl být *sloupec výsledku*). V diagramu například vidíme query subject *Order*, který se skládá z query items *Order number* a *Order detail code*. Vidíme, že query subject *Order* se skládá z query items *Order number* a *Order detail code*.

Query subject může být definován SQL dotazem do relační databáze, vloženou procedurou (pak reprezentuje výsledek vložené procedury) nebo jako dotaz využívající query items (sloupce výsledku) z jiných výsledků dotazu (query subjects) nadefinovaných v datovém modelu. Objekty výsledků dotazu by měly vést na logické celky, které dávají ucelený smysl. Například query subject Sales reprezentuje realizovanou objednávku.



Obrázek 1.2: Ukázka query subjects v modelu



Obrázek 1.3: Ukázka hierarchie v modelu

Diagram 1.2 vzhledem připomíná ER diagram relační databáze – například tím, že mezi entitami diagramu jsou znázorněny vztahy včetně jejich kardinality. Je důležité si uvědomit, že datový model na úrovni query subjects není nutně přímým odrazem relačního datového modelu. Jednotlivé query subjects jsou celky logické, nikoliv fyzické úrovně a jeden query subject může být namapován na několik tabulek v databázi. Naopak jedna tabulka může sloužit k definici více query subjects modelu. Stejně tak query items nereprezentují pouze sloupce v tabulce databáze, query item může vznikat výpočtem z hodnot několika sloupců databáze. Na úrovni modelu ale tento výpočet tvoří atomickou jednotku.

Objekty datového modelu jsou uspořádány hierarchicky. Query items jsou na nejnižší úrovni. Ty se sdružují jednak do query subjects, ale také do dalších typů objektů. Na obrázku 1.3 vidíme *hierarchii Time* reprezentující čas. Ta se skládá z jednotlivých úrovní hierarchie *(levels)*, vidíme zde úrovně *Time*, *Year*, *Quarter*, *Month* a *Day*, přičemž vidíme, že postupně klesá úroveň granularity – čas se skládá z roků, rok se skládá z kvartálů, atd. U úrovní hierarchie *Month* a *Day* vidíme query items, ze kterých se tyto úrovně skládají.

Objekty typu *výpočet (calculation)* vytváří nové datové položky z existujících. Například výpočet *Order value* je definován následujícím vzorcem: `total ([gosales].[Sales].[Quantity] * [gosales].[Sales].[Unit sale price] for [gosales].[Sales].[Order number])`. Vystupují zde query items *Quantity*, *Unit sale price* a *Order number*.

Filtry (filters) lze aplikovat na query subject nebo na dimenzi a provádí restrikci množiny hodnot query subjectu nebo dimenze podle definice filtru. Filtr *Returns - unsatisfactory or defective* je definován: `[Business view].[Return reason].[Return reason code] in (1,5)` a omezuje query subject *Return reason* na ty entity, jejichž atribut *Return reason code* (což je query item) nabývá hodnot 1 nebo 5.

Dále se v datovém modelu vyskytují *namespaces* (jmenné prostory), které odpovídají adresářům, do nichž lze sdružovat ostatní objekty. Posloupnost namespaces, v nichž je objekt vnořen, tvoří cestu k objektu. *Odkazy (shortcuts)* umožňují ukazovat na jiné objekty datového modelu (stejně jako soft link/zástupce v operačních systémech). Častým užitím namespaces v kombinaci s odkazy je vytvoření více odlišných pohledů na stejné objekty datového modelu.

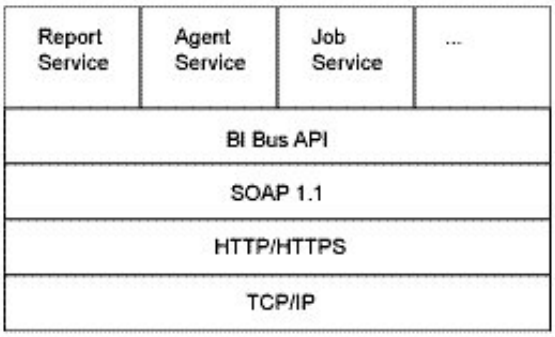
Reporty se neodkazují přímo na datový model, ale na *balíček modelu* (*model package*), které datový model definuje. Balíčky modelu definují podmnožinu objektů datového modelu určenou k publikaci – v reportu lze využít pouze objekty modelu z právě jednoho balíčku. Datový model přitom typicky definuje více balíčků.

1.3 Aplikační rozhraní platformy IBM Cognos

Po obecném přehledu platformy IBM Cognos v předcházející části kapitoly přejdeme v následující části do detailu aplikačního rozhraní platformy. Pro pochopení toho, jak je API strukturováno, je nutný stručný nástin architektury platformy. Následuje nejprve přehled služeb platformy a poté jejich detailní popis z hlediska programátora – uživatele aplikačního rozhraní těchto služeb.

1.3.1 Architektura platformy

Platforma IBM Cognos je postaven ve stylu servisně orientované architektury (SOA). Jednotlivé komponenty platformy (v terminologii SOA označované jako *služby*) spolu komunikují pomocí webových služeb přes rozšířený protokol SOAP. Služby IBM Cognos spolu vzájemně komunikují pomocí rozhraní, které se v terminologii Cognos nazývá *BI Bus API*. Toto rozhraní může pro komunikaci s platformou využívat i aplikační vývojář, ale typicky tak nedělá na přímo zasíláním SOAP zpráv.



Obrázek 1.4: Architektura BI Bus API (Zdroj: [2])

1.3.2 IBM Cognos Software Development Kit

Platforma IBM Cognos poskytuje jako součást SDK knihovny pro jazyk Java a platformu Microsoft .NET. Tyto knihovny umožňují služby platformy volat jako metody nad objekty v příslušném programovacím jazyce. Konzument API tedy nemusí přímo komunikovat s BI Bus API pomocí zpráv protokolu SOAP. Fakt, že platforma je postavena na servisně orientované architektuře pro vývojáře má pouze ten důsledek, že objekty API v knihovně odpovídají svými názvy a rozdělením odpovědností službám, které definuje SOA architektura platformy. Toto ilustruje obrázek 1.4. Například službě Report Service odpovídá objekt API `ReportService_PortType`.

SDK dále obsahuje definice pro popis jazyků XML, používaných jednotlivými API. Definice jsou ve standardním jazyce XML Schema (XSD).

V rámci SDK jsou distribuovány knihovny API a dále knihovny, na kterých knihovny API závisí.

1.3.3 Přehled služeb platformy

Jak již bylo zmíněno, celá platforma je rozdělena do jednotlivých služeb. Ze všech služeb platformy (které uvádí [2]) nás zajímají pouze ty, které poskytují funkcionalitu využitelnou pro potřeby loaderu. Například nás nezajímá služba *Human Task Service*, pomocí které lze v platformě spravovat notifikace pro uživatele a akce, které má uživatelé provést.

1.3.4 Content Manager Service

Tato služba umožňuje spravovat tzv. *Content Store*, což je relační databáze sloužící pro persistenci dat celé platformy. Služba Content Manager Service umožňuje z Content Store získávat objekty pokládáním dotazů, vkládat nové objekty, existující objekty aktualizovat, přesouvat, kopírovat a mazat.

Náš konektor na vstupu dostane cestu k reportům, které má zpracovat. Bude také potřebovat získávat další objekty, které s reportem souvisí. Pro oba tyto účely využijeme službu Content Manager Service.

1.3.4.1 Search path

Dotazy se pokládají pomocí *search path* výrazů. Syntaxe je ekvivalentní syntaxi dotazovacího jazyka XPath, ale na rozdíl od XPath se nedotazujeme na uzly XML dokumentu, ale na objekty platformy IBM Cognos v Content store. Například takto vypadá search path výraz, který vede na demonstrační report *2011 Sales Summary*: `/content/folder[@name='Packages']/package[@name='GO Sales (analysis)']/folder[@name='Report Studio Report Samples']/report[@name='2011 Sales Summary']`. Platforma každému objektu Content store přiřazuje jeho search path. Vzhledem k možnostem jazyka XPath lze sestrojit více výrazů, jejichž vyhodnocení povede na jeden objekt.

1.3.5 Metadata Service

Tato služba umožňuje dotazování nad nepublikovanými modely komponenty Framework Manager a také modifikaci těchto modelů.

Služba dále umožňuje získávat informaci, které objekty datového modelu jsou používány datovými položkami (*data items*) reportu jako jejich datové zdroje. Obdobně lze u objektů datového modelu zjistit jejich datové zdroje (jimiž jsou jiné objekty datového model). Tato informace se v terminologii BI nazývá *data lineage* a je jednou z podstatných informací, kterou chceme v katalogu reportů prezentovat.

1.3.6 Report Service

Tato služba slouží k *spouštění* reportů, což typicky nastává, když uživatel přes webového rozhraní Cognos zadá požadavek k otevření reportu. To je primárně interní funkcionalita IBM Cognos a pro potřeby konektoru není zajímavá.

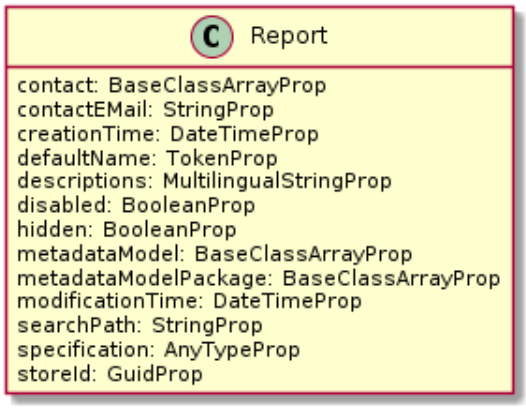
Služba dále slouží k získání data lineage reportu. Na vstupu dodáme XML dokument specifikující report pomocí jeho search path a na výstupu dostaneme specifikaci, která popisuje data lineage jednotlivých datových položek dotazovaného reportu.

Report Service dále umožňuje upravovat definice reportů.

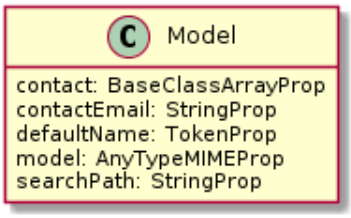
1.4 Java API platformy Cognos

1.4.1 Objekty report, model a package

Z objektů platformy IBM Cognos nás primárně zajímají reporty. Report je na úrovni API reprezentován objektem `Report`. Report využívá data z právě



Obrázek 1.5: Podstatné atributy objektu Report



Obrázek 1.6: Podstatné atributy objektu Model

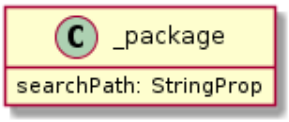
jednoho balíčku datového modelu (objekt `_package`). Balíček je součástí datového modelu (objekt `Model`).

N obrázku 1.5 je třídní diagram objektu `Report`. Zobrazuje atributy tohoto objektu, které jsou relevantní z hlediska loaderu. Mezi tyto atributy patří jednak metadata reportu a druhak data, která potřebujeme pro další zpracování loaderem. Mezi metadaty reportu jsme vybraly ta, která chceme katalogizovat. Patří k nim například datum poslední změny reportu (atribut `modificationTime`).

Do kategorie dat pro další zpracování loaderem patří XML specifikace reportu (atribut `specification`). Specifikaci reportu potřebujeme pro získání řady informací, které nelze získat na úrovni objektu API, jedná se zejména o strukturu reportu. XML specifikaci reportu se podrobněji věnuje část 3.10. Dále do kategorie dat pro další zpracování patří atributy `metadataModelPackage` a `metadataModel`. Tyto atributy odkazují na objekty `_package` a `Model`, které objekty odpovídají balíčku datového modelu,respektive celému datovému modelu.

Diagram 1.5 zobrazuje podstatné atributy objektu `Model`.

Zásadní je pro nás atribut `model`, který obsahuje XML dokument se specifikací modelu. Specifikaci modelu potřebujeme pro získání informací o objektech modelu. Zpracování XML specifikace modelu se blíže věnuje část 3.6.



Obrázek 1.7: Podstatné atributy objektu _package

Objekt `_package` reprezentující balíček modelu (viz 1.7) obsahuje převážně atributy reprezentující informace technického charakteru, které nejsou z pohledu katalogu reportů zajímavé. Podstatný je pouze atribut `searchPath`, která vyjadřuje cestu k balíčku v rámci Content store (viz část 1.3.4.1).

Atributy zmíněných objektů API jsou instancemi tříd, odvozených od typu `BaseProp`. Diagram 1.8 ilustruje hierarchii těchto typů. Všechny Prop typy obsahují atribut a obsahují metodu `getValue` pro získání hodnoty tohoto atributu (s výjimkou typu `BooleanProp`, jehož metoda má název `isValue`).

Řada atributů objektů API je abstraktního typu `BaseProp` (jako v případě atributu `contact` v třídě `Report`). V takovém případě při práci s tímto atributem musíme provádět kontrolu na konkrétní typ objektu a následné přetypování. Toto práci s API komplikuje.

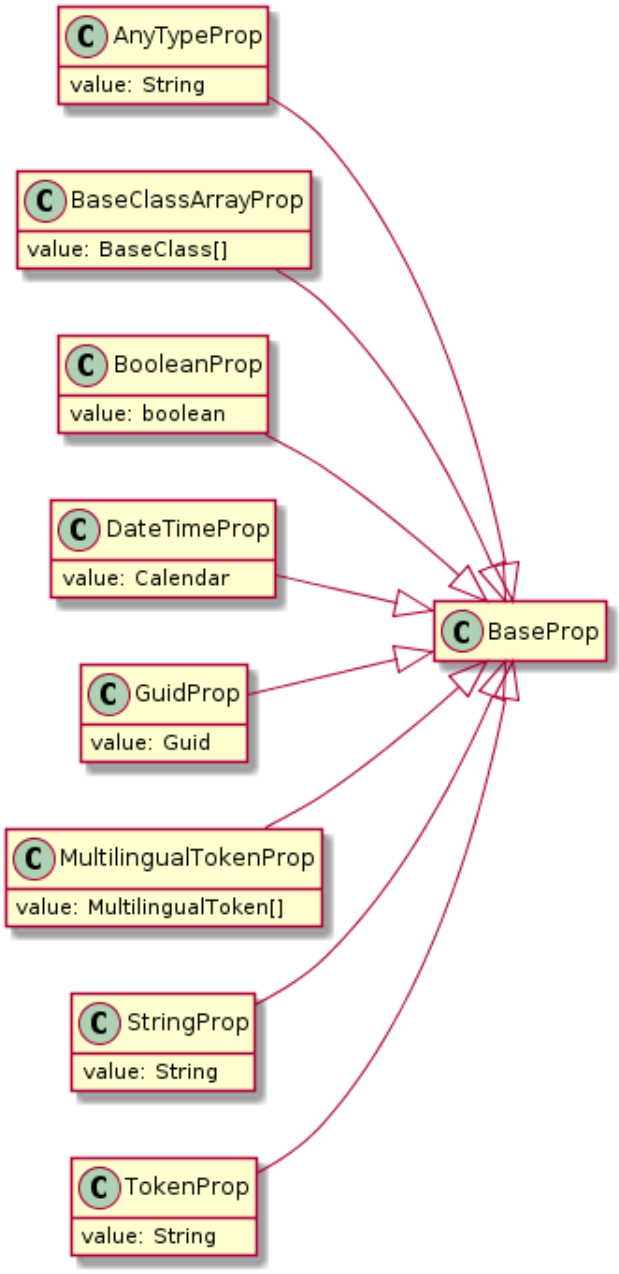
Objekty `Report`, `Model` i `_package` se získávají pomocí služby Content Manager Service.

1.4.2 Content Manager Service

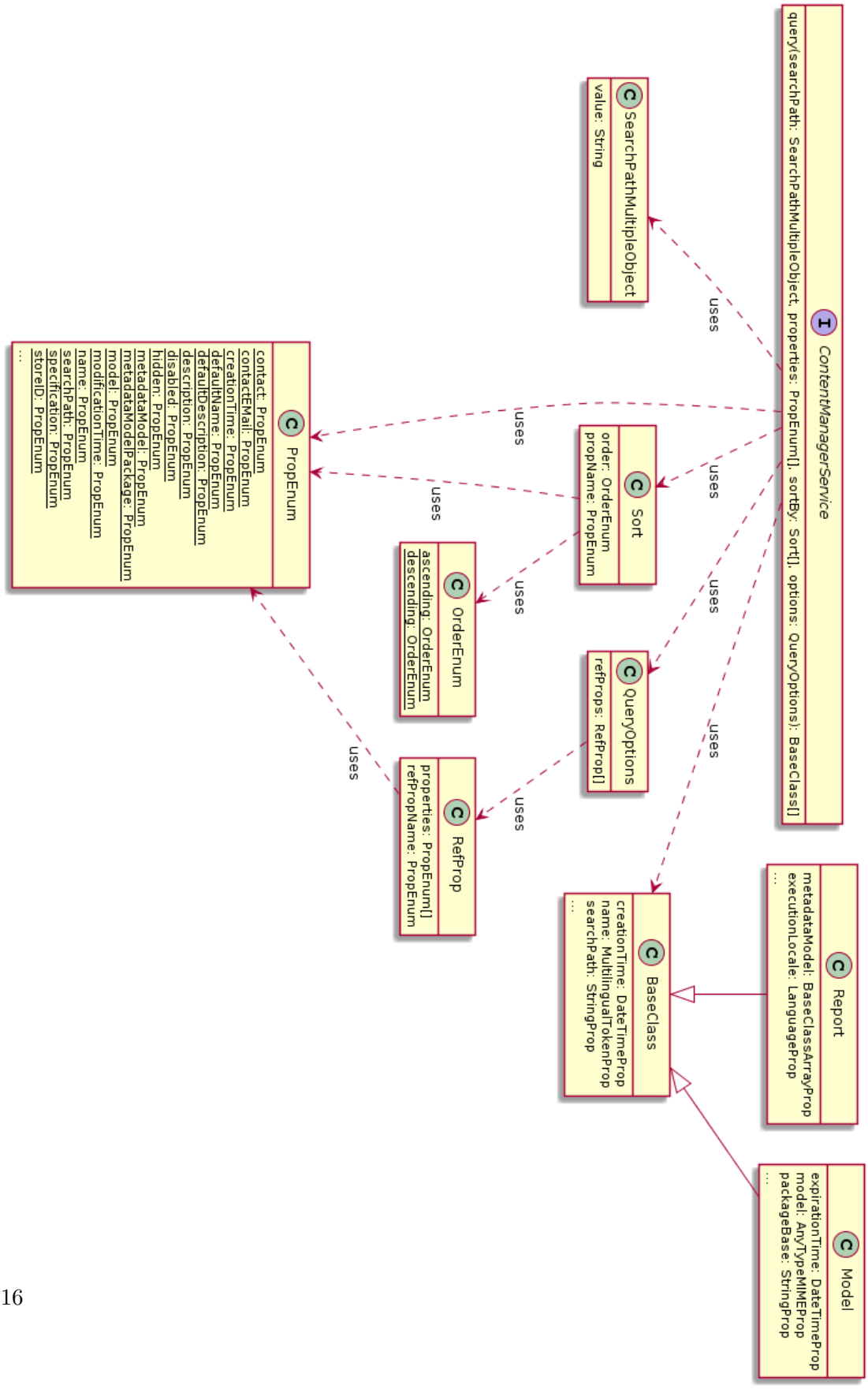
Obázek 1.9 zobrazuje třídní diagram API Content Manager Service zjednodušený na ty objekty a metody, které potřebujeme pro implementaci loaderu. Vstupním bodem do této služby je objekt ContentManagerService. Obsahuje 2 metody, které jsou pro nás podstatné – `query` a `queryMultiple`. Parametry metod jsou netriviální, proto je v diagramu nastíněna jejich struktura.

1.4.2.1 Metoda query

Metoda `query` slouží k získání objektů platformy (např. objekt `Report` nebo objekt `FrameworkManagerModel`) z Content store. Metoda vrací pole objektů typu `BaseClass`, z nichž jsou děděním odvozeny další objekty jako `Report` a `Model`. Hodnoty proměnných objektu `BaseClass`, které chceme posléze vyčíst, musíme předem specifikovat při volání metody, jinak nabývají hodnoty `null`. Tato technika, kdy klient API je nucen předem specifikovat záměr s výsledkem slouží k tomu, aby nedocházelo k přenosu zbytečně velkých objektů. Ve fragmentu 1.1 je příklad dotazu na objekt report s požadovanou hodnotou proměnné `description`.



Obrázek 1.8: Hierarchie typů odvozených od BaseProp



Obrázek 1.9: API Content Manager Service - metoda query

Fragment 1.1: Volání metody query služby Content Manager Service

```
ContentManagerService_PortType cmService = ...
String reportPath = "//report[@name='2011 Sales Summary']";
SearchPath searchPath = new SearchPathMultipleObject(reportPath)
PropEnum[] props = { PropEnum.description };
BaseClass[] result = cmService.query(searchPath, props, new Sort[0], new QueryOptions());
```

Rozebereme podrobněji jednotlivé parametry metody `query`.

Parametr `searchPath` udává cestu k objektu pomocí search path. Tento parametr je typu `SearchPathMultipleObject`, což je pouze obálka na řetězec obsahující search path výraz.

V parametru `properties` specifikujeme polem objektů třídy `PropEnum`, které atributy objektu vráceným metodou `query` požadujeme neprázdné. Z diagramu 1.9 vidíme, že jednotlivé instance `PropEnum` jsou k dispozici jako statické proměnné ve třídě `PropEnum` (instancí existuje řádově více, v diagramu jsou zobrazeny pouze ty, které loader využívá).

Parametr `sortBy` typu `Sort` slouží ke specifikaci požadovaného řazení objektů ve výsledku. Řadit lze vzestupně nebo sestupně (směr řazení vyjadřuje objekt `OrderEnum`) a řadí se podle zvoleného atributu výsledných objektů. Atribut se specifikuje instancí třídy `PropEnum`. Pokud nemá být výsledek seřazen, lze předat instanci objektu `Sort` vytvořenou voláním konstruktoru bez parametrů, viz fragment 1.1.

Poslední parametr `options` typu `QueryOptions` specifikuje dodatečné parametry dotazu, jako je například limit na počet výsledků. Tyto dodatečné parametry jsou pro přehlednost z diagramu 1.9 vynechány. Parametr `options` má další důležitou funkci, a to specifikaci tzv. *nested* atributů. Výše byl popsán mechanismus, kterým je nutné předem specifikovat, které atributy výsledku se mají naplnit hodnotou. Toto se komplikuje v případě, že jeden nebo více z těchto atributů je objektového typu. V takovém případě se zde stejný problém přesouvá na další hladinu – API nutí volajícího specifikovat požadované atributy i tohoto atributu. Ve fragmentu 1.2 takto specifikujeme, že chceme získat z objektu report jeho atribut `metadataModel` a z tohoto atributu (typu `metadataModel`) požadujeme atributy `defaultName` a `searchPath`.

Fragment 1.2: Specifikace nested properties

```
ContentManagerService_PortType cmService = ...
String reportPath = "//report[@name='2011 Sales Summary']";
SearchPath searchPath = new SearchPathMultipleObject(reportPath);
PropEnum[] topLevelProps = { PropEnum.metadataModel };
PropEnum[] nestedProps = { PropEnum.defaultName, PropEnum.searchPath };
RefProp[] refProps = { new RefProp(nestedProps, PropEnum.metadataModel) };
QueryOptions queryOpts = new QueryOptions();
queryOpts.setRefProps(refProps);
BaseClass[] result = cmService.query(searchPath, topLevelProps, new Sort[0], queryOpts);
```

1.4.2.2 Metoda queryMultiple

Pomocí metody `queryMultiple` lze zadat najednou několik dotazů, které by při použití metody `query` vedly na opakované volání metody. Jak je vidět z diagramu 1.10, metoda přijímá parametr `requests`, který je typu pole objektů `QueryRequest`. Objekt `QueryRequest` zabaluje dohromady všechny parametry, které používá předchozí metoda `query` s drobným rozdílem, že search path výraz se zadává přímo řetězcem bez obalovacího objektu. Typem návratové hodnoty metody `queryMultiple` je `QueryReply`, přičemž jedna instance `QueryReply` zabaluje odpověď odpovídající jednomu `QueryRequest` požadavku.

1.4.3 Report Service

1.4.3.1 Metoda runSpecification

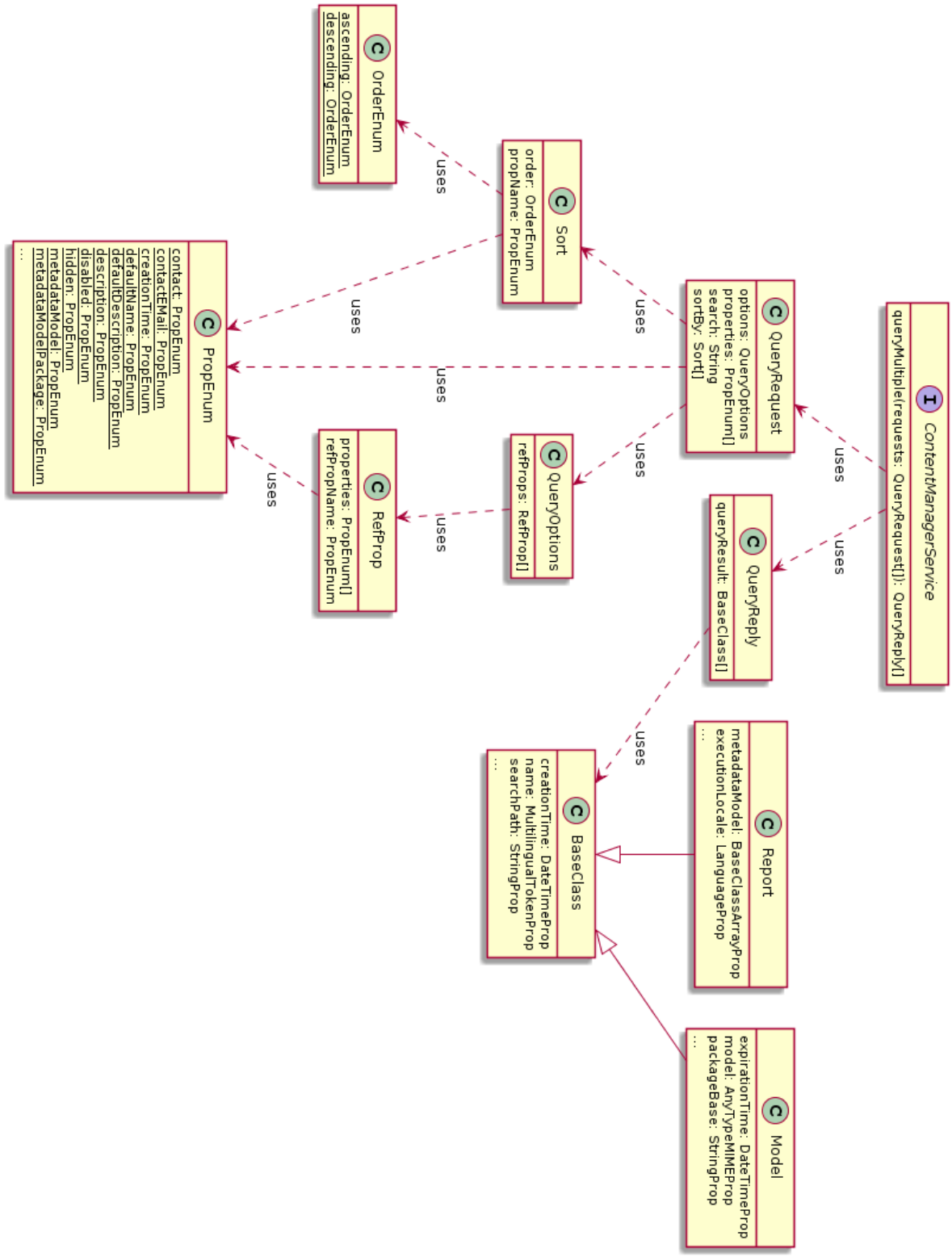
Pomocí metody `runSpecification` lze spustit specifikaci. Specifikace je XML dokument, který popisuje objekt platformy. Lze takto „spustit“ report dodáním jeho XML specifikace nebo zaslat požadavek na data lineage reportu (jak jsme zmínili v sekci 1.3.6).

Proces získání lineage reportu je v [2] popsán takto:

> „Beginning in IBM® Cognos® Business Intelligence V10.1.0, you must issue lineage requests for report data against reportService or batchReportService to obtain querySet data, which you must then deliver to metadataService in a subsequent lineage request.“

Zasláním požadavku na lineage reportu do Report Service obdržíme XML dokument, popisující datové položky reportu (neobsahuje informaci o lineage datových položek). Na základě jeho zpracování jsme následně schopni vytvořit druhý požadavek na lineage datových položek reportu, který použijeme pro volání Metadata Service.

Fragment 1.3 představuje ukázku odpovědi Report Service na požadavek na lineage reportu. Vidíme, že element `<queries>` se odkazuje na XML namespace `http://developer.cognos.com/schemas/report/9.0/`, což je namespace odpovídající specifikaci reportu. Při srovnání s ukázkou XML dokumentu specifikace stejného reportu (obsažené v objektu `Report` získaného z Content Manager Service) ve fragmentu 1.4 zjistíme, že zde se nachází element `<queries>` se shodným obsahem – informace o queries reportu jsou identické v obou dokumentech.



Obrázek 1.10: API Content Manager Service - metoda queryMultiple

Fragment 1.3: XML dokument lineage reportu

```
<reportLineageResponse>
   <lineageResponse>
      <object id="[Report]">
         <name>Order Invoices - Donald Chow, Sales Person</name>
         <type>baseReport</type>
         <property name="objectType" displayName="Type">Report</property>
         <childRef>/content/folder[@name='Packages']/package[@name='GO Sales (query)']</childRef>
         <property name="description" displayName="Description">Generates invoices of sales by Donald Chow.</property>
         <property name="owner" displayName="Owner">Anonymous</property>
      </object>
   </lineageResponse>
   <querySet expressionLocale="en-ca">
      <modelPath>/content/folder[@name='Packages']/package[@name='GO Sales (query)']/model[@name='model']</modelPath>
      <queries xmlns="http://developer.cognos.com/schemas/report/9.0/">
         <query name="Query - Order Invoice">
            <source>
               <model />
            </source>
            <selection>
               <dataItem aggregate="none" name="Order number" rollupAggregate="none" sort="descending">
                  <expression>[Sales (query)].[Order].[Order number]</expression>
               </dataItem>
              <dataItem aggregate="none" name="Order method">
                 <expression>[Sales (query)].[Order method].[Order method type]</expression>
              </dataItem>
               ...
```

Fragment 1.4: XML specifikace reportu – queries

```
<report xmlns="http://developer.cognos.com/schemas/report/9.0/" expressionLocale="en-ca">
    ...
    <queries>
        <query name="Query - Order Invoice">
            <source>
                <model/>
            </source>
            <selection>
                <dataItem aggregate="none" name="Order number" rollupAggregate="none" sort="descending">
                    <expression>[Sales (query)].[Order].[Order number]</expression>
                </dataItem>
                <dataItem aggregate="none" name="Order method">
                    <expression>[Sales (query)].[Order method].[Order method type]</expression>
                </dataItem>
               ...
```

XML lineage reportu obsahuje oproti XML specifikaci reportu navíc element `<queryResultDefinitions>`, který konkretizuje použití jednotlivých datových položek query. Tato informace je pro potřeby našeho loaderu už příliš detailní a v metadata katalogu ji nevyužijeme.

Protože z lineage reportu nás zajímá pouze uzel, tvořený elementem `<queries>`, který je obsažen i v XML specifikaci reportu, nedodává nám dokument lineage reportu žádnou novou informaci. Specifikaci reportu potřebujeme zpracovávat pro získání informací o struktuře reportu. XML dokument s požadavkem na lineage datových položek reportu jsme tedy schopni vygenerovat na základě specifikace reportu.

Při implementaci konektoru tedy nebudeme API Report Service potřebovat.

1.4.4 Metadata Service

Metadata Service obsahuje stejně jako Report Service metodu `runSpecification`, která slouží ke spouštění specifikací. V případě Metadata Service se jedná o specifikace požadavků na lineage objektů datového modelu.

1.4.5 Lineage XML dokumenty

Metoda `runSpecification` služby Metadata Service přijímá na vstupu XML dokument požadavku na lineage a na výstupu vrací XML dokument se specifikací lineage. V implementaci konektoru využíváme tuto metodu k získání informace o lineage datových položek reportu a následně lineage objektů datového modelu. XML dokumentům lineage se podrobněji věnuje kapitola 3.

1.5 Analytický model objektů platformy Cognos

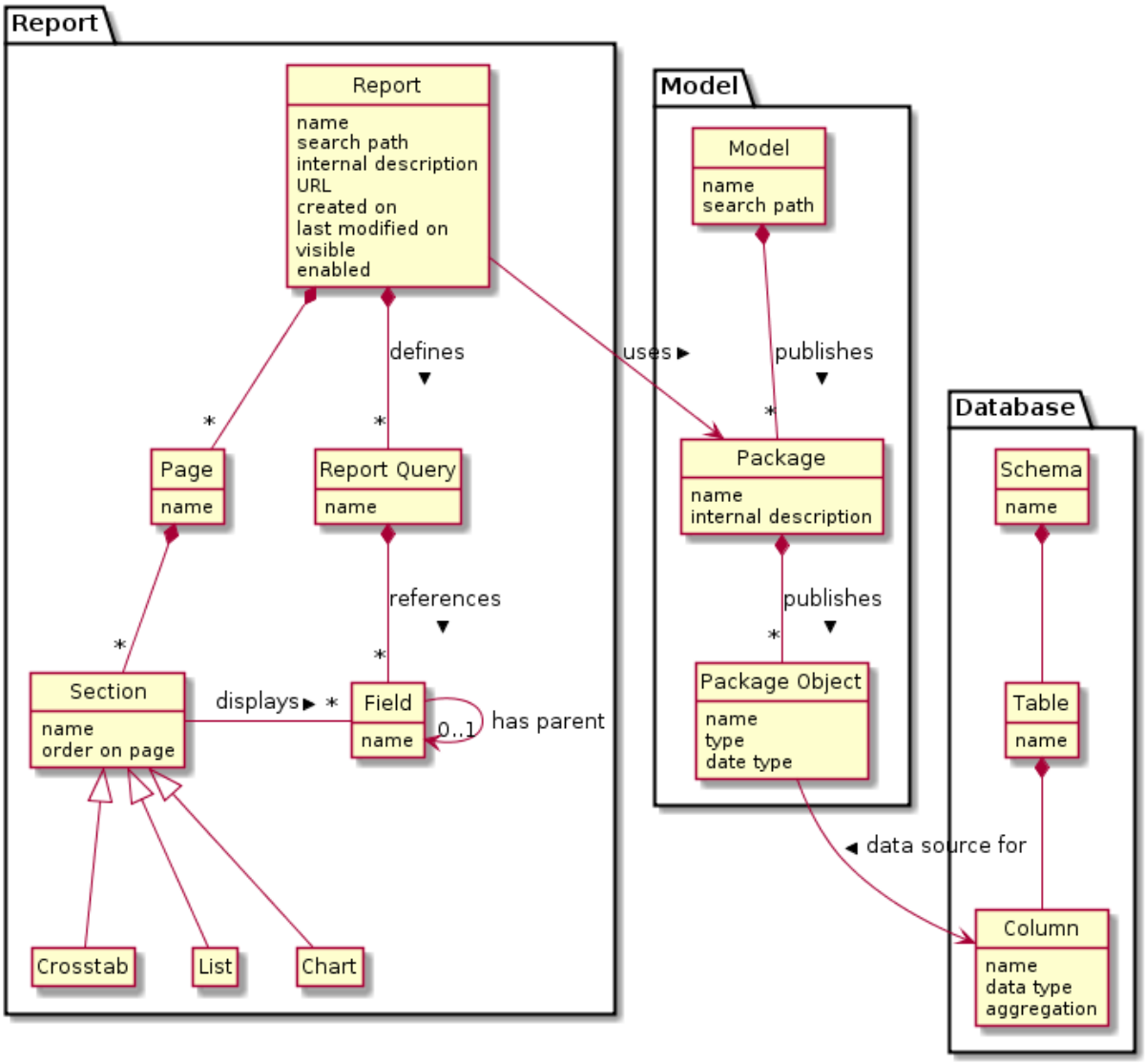
V sekcích 1.1 a 1.2 jsme uvedli, jaké objekty platformy IBM Cognos rozlišujeme. Z pohledu katalogu reportů nás u reportu zajímají informace o jeho struktuře, dále pak informace o datovém modelu, ze kterého report čerpá data a nakonec informace o lineage položek reportu.

V sekci 1.4.1 jsme si přiblížili, jaká metadata lze získat o reportech a o datovém modelu.

Kombinací těchto informací dostaneme analytický model objektů platformy a jejich vlastností, které nás zajímají pro účely katalogizace. Diagram 1.11 představuje tento analytický model. Jsou zde vyznačeny tři vrstvy, do kterých rozdělujeme entity modelu. Je to vrstva reportu, vrstva datového modelu a fyzická vrstva databáze.

Databázovou vrstvu jsme v textu doposud nezmínili. V logickém datovém modelu platformy Cognos se vyskytují objekty *zdroje dat* (*Data Sources*), které obsahují informaci pouze o názvu databáze (resp. názvu schématu). V kapitole 3 ale uvidíme, že informaci o struktuře databáze jsme do značné míry schopni odvodit z data lineage některých objektů datového modelu.

Entity v analytickém modelu 1.11 definují některé vlastnosti, u kterých nemusí v tento moment zřejmý jejich účel. V následujících kapitolách budou tyto části postupně objasněny.



Obrázek 1.11: Analytický model – objekty platformy Cognos, které chceme katalogizovat

Kapitola 2

Platforma XF3

V kapitole 1 jsme identifikovali typy objektů platformy IBM Cognos, které nás z zajímají z pohledu katalogu reportů (report, model, balíček modelu) a vlastnosti, které u nich chceme evidovat. V této kapitole se seznámíme s aplikační platformou, na které budeme realizovat prezentační část metadata katalogu.

2.1 Přehled vlastností platformy XF3

XF3 je platforma pro deklarativní vytváření formulářových aplikací, vyvinutá firmou Semanta s.r.o. Platforma XF3 je naše účely vhodná zejména proto, že umožňuje deklarativně definovat strukturu metadata katalogu objektově orientovaným způsobem.

Pomocí speciálního XML jazyka platformy můžeme nadefinovat „třídu“ report a její „pole“ jméno, popis atd. V terminologii XF3 však třídu nazýváme *aplikace*. Jednotlivé instance této třídy (v případě katalogu reportů například jednotlivé reporty) nazýváme anglickým slovem *entry*. Jednotlivá pole aplikací (a zároveň instancí – entries) se nazývají *fields*. České ekvivalenty pro tyto pojmy nezavádíme.

Na obrázku A.3 vidíme ukázku nasazení platformy XF3 s otevřenou úvodní stranou. Úvodní strana je definována jako XF3 aplikace, speciální v tom, že od ní existuje jenom jedno entry. Na obrázku vidíme, že aplikace pro úvodní stranu obsahuje přehled dalších entries, které představují podstatné XF3 aplikace. Dále je zde seznam naposledy aktualizovaných entries a blog. Všechny tyto prvky jsou realizovány jako fields v rámci aplikace úvodní strana.

Pro naše účely jsou z hlediska definice aplikací zásadní následující možnosti, které platforma XF3 nabízí:

- Hierarchické vnořování entries. XML jazyk pro definici aplikací umožňuje vyjádřit, že reporty obsahují stránky, stránky obsahují bloky, a bloky obsahují datové položky. Lze hierarchicky vnořovat i entry stejného typu. V našem případě může entry aplikace *Data Item* (odpovídající datové položce na reportu) mít několik potomků aplikace *Data Item*. Tento vztah modeluje situaci u kontingenčních tabulek, které umožňují hierarchii datových položek.

- Další vztahy mezi entries. Jeden typ entry může vystupovat v několika vztazích s entry dalších XF3 aplikací. Můžeme například vyjádřit, že entry aplikace *Report* může mít vztah typu „má zdroj dat“ s několika entry aplikace *Table*. Platforma také umožňuje snadno dodefinovat opačný směr vztahu. U výše zmíněného vztahu „má zdroj dat“ nás zajímá i opačný směr vztahu – „je zdrojem dat“.

Vedle samotné definice struktur platforma XF3 umožňuje vytváření entries na základě dat načtených z externího zdroje dat (primárně relační databáze,

další možné typy zdrojů dat jsou LDAP a aplikace Atlassian Confluence). Vytváření entries probíhá během *loadu* spuštěním *nahrávací úlohy*, která je nadefinována v dalším XML jazyce platformy XF3. Definice nahrávací úlohy se skládá ze dvou částí. První část představují SQL dotazy pro získání dat z relační databáze (nebo jiný způsob dotazování na data v případě nerelačního zdroje dat). Druhá část definuje mapování dat získaných v první části na fieldy aplikací XF3.

Jedna definice nahrávací úlohy může definovat mapování dat pro celou sadu aplikací. Typicky existuje jedna úloha pro jeden zdroj dat (jako jsou například informace o reportech IBM Cognos získané loaderem), v jehož rámci je definováno mapování dat ze zdroje na více aplikací (např. report a objekt modelu).

V našem případě je zdrojem dat pro nahrávací úlohu schéma v relační databázi, do kterého zapisuje loader záznamy o objektech platformy IBM Cognos. Jedním XML souborem s definicí úlohy popíšeme, jak z dat v relační databáze vytvořit v XF3 kolekce entries reportů, jejich sekcí, objektů datového modelu a dalších aplikací.

Další zajímavou funkcionalitou XF3 je možnost kombinovat údaje získané nahráním z externího zdroje s ruční editací přímo ve webovém rozhraní platformy XF3. Některé informace o objektech Cognos BI nelze vyčíst přímo z platformy jako takové. Jedná se například o uživatelské vysvětlení jednotlivých sekcí reportu. Tyto vlastnosti může doplnit uživatel metadata katalogu úpravou přímo v XF3. Pokud se změní report v Cognos BI, je nutné provést nový load. Je zásadní, aby v takovém případě nedošlo ke ztrátě těchto ručně dodaných informací. Platforma XF3 umí toto na úrovni nahrávacích úloh zajistit.

Platforma XF3 dále nabízí umožňuje základě definic aplikací automaticky generovat diagramy vztahů mezi entries. Komponenta pro vizualizaci vztahů se nazývá *Nexus* a na obrázku A.4 je její ukázka. Z pohledu uživatele katalogu reportů je tato možnost přínosná, protože mezi identifikovanými objekty IBM Cognos se objevuje řada vazeb. Z tohoto pohledu jsou zajímavé zejména vazby mezi reporty a objekty datového modelu a vazby mezi objekty datového modelu a sloupci tabulek relační databáze (viz 1.11). Tyto vztahy představují tzv. *lineage*, popis „přítoku“ dat do reportu.

Pro zajištění dostupnosti dokumentace přímo z aplikací platformy IBM Cognos můžeme využít funkcionality, kterou nabízí komponenta Air platformy XF3. Komponenta Air umožňuje pomocí JavaScriptu dopravit do libovolné webové aplikace kontextové informace z platformy XF3. Air funguje způsobem, kdy uživatel otevřením záložky v prohlížeči spustí skript, který vezme URL aktuální stránky v prohlížeči a odešle dotaz do XF3 instance. Tam proběhne zpracování požadavku a vyhodnocení, zda k požadované URL existuje odpovídající entry v některé z aplikací.

Pokud k URL existuje entry, je uživateli zobrazeno přímo v rámci stránky pomocí iframe, který Air do stránky vloží. Uživatel může s entry manipulovat

způsobem, jako kdyby se nacházel přímo v platformě XF3. Tato situace je vidět na obrázku A.2.

Pokud entry k dané URL doposud neexistuje, Air nabídne uživateli entry vytvořit a popsat opět přímo ze stránky, na které se nachází.

2.2 Ukázková aplikace

Pro účely implementace katalogu reportů jsou klíčové dvě vlastnosti platformy: definice aplikací a definice nahrávací úlohy pro naplnění aplikací z externího zdroje – relační databáze. Provedeme demonstraci těchto vlastností na nějakém obecně pochopitelném příkladě, abychom současně neřešili komplexitu platformy XF3 a (čtenáři možná méně známého) světa reportingu.

Ve zbytku kapitoly vytvoříme jednoduchou aplikaci pro evidenci zaměstnanců a vytvoříme k ní definici nahrávací úlohy pro načtení zaměstnanců z relační databáze. Jako zdroj dat pro load použijeme vzorovou databázi, používanou v dokumentaci databáze MySQL.

2.2.1 Schéma DB – zdroje dat

Využijeme databázi *test_db*, která je distribuována pod licencí Creative Commons Attribution-Share Alike a je dostupná z GitHub repozitáře [3]. Databáze obsahuje schéma `employees`, ve kterém jsou tabulky reprezentující evidenci zaměstnanců fiktivní firmy. V rámci loadu dat se omezíme na podmnožinu schématu, kterou zobrazuje diagram 2.1. Schéma obsahuje tabulku `employees` se zaměstnanci, tabulku `departments` s odděleními firmy a vazební tabulku `dept_emp`, reprezentující zařazení zaměstnanců do oddělení.

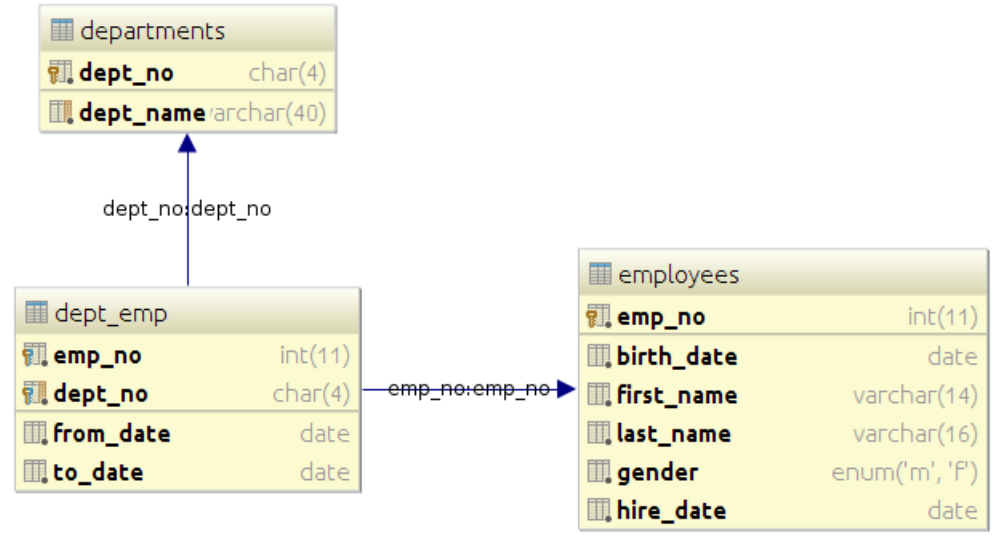
2.2.2 Definice ukázkových aplikací

V naší evidenci zaměstnanců se budou vyskytovat aplikace zaměstnanec (*Employee*) a oddělení (*Department*). U zaměstnance budeme evidovat jméno, datum narození a zařazení do oddělení. U oddělení budeme evidovat jeho název. Mezi těmito aplikacemi budeme evidovat vztah vyjadřující, že zaměstnanec patří do oddělení.

2.2.2.1 Aplikace department

Fragment 2.1: Definice aplikace department

```
<?xml version="1.0" encoding="UTF-8"?>
<xforms-apps xmlns="http://semanta.cz/schema/xf3">
    <xforms-app>
        <xid>department</xid>
        <entry-name>Department</entry-name>
        <structure>
            <row>
                <column>
                    <width>12</width>
```



Obrázek 2.1: Podmnožina DB schématu sloužící jako zdroj dat

```
                <group>
                    <id>basic</id>
                    <fields>
                        <field>
                            <code>entryName</code>
                            <name>Department Name</name>
                            <type>
                                <plain/>
                            </type>
                        </field>
                        <field>
                            <code>hasEmployees</code>
                            <name>Employees</name>
                            <type>
                                <relations>
                                    <generator-code>xf3search:${text} (type:employee)</generator-code>
                                    <relation-type>has-employee</relation-type>
                                    <outgoing-relation>true</outgoing-relation>
                                    <display-as>table</display-as>
                                    <multiple>true</multiple>
                                </relations>
                            </type>
                        </field>
                    </fields>
                </group>
            </column>
        </row>
        <list>
            <fieldCode>entryName</fieldCode>
        </list>
    </structure>
    <listeners/>
</xforms-app>
</xforms-apps>
```

2.1 definuje entry aplikace *Department*. Všechny objekty platformy XF3 mají svůj unikátní identifikátor, *xid*. U většiny typů entit platformy je přiřazován platformou automaticky, ale u aplikace je definován manuálně v elementu `<xid>`. V našem případě je jeho hodnota `department`.

Element `<entry-name>` definuje název typu entry, což je metadatová položka, kterou platforma XF3 využívá v situacích, kdy se mluví o třídě entries jedné aplikace. Typ entry naší aplikace pojmenujeme *Department*.

Definice polí entry (v terminologii XF3 *fields*) není oddělena od definice jejich rozložení na stránce s prezentací entry. Element `<structure>` definuje řádky – element `<row>`, ve kterých se definují sloupce – element `<column>`. Tyto elementy ovlivňují podobu prezentace entry. V našem případě použijeme nejjednodušší možný layout – jeden řádek s jedním sloupcem, ve kterém bude jedna skupina fieldů.

Sloupce nabývají šířky (element `<width>`) od 1 do 12. Hodnota 1 odpovídá šířce $\frac{1}{12}$ dostupného prostoru. Maximální hodnota 12 odpovídá plné šířce.

Skupinu definuje element `<group>` a má svůj identifikátor `<id>` pro interní použití platformou. V rámci skupiny je element `<fields>`, ve kterém definujeme jednotlivé pole (fieldy) entry, kterým odpovídá element `<field>`.

Field má svůj interní kód (`<code>`) a název, který je zobrazován uživatelům (`<name>`). Každý field má typ (`<type>`), který určuje datový typ hodnoty v něm uložené a vizuální podobu políčka pro zobrazení a editaci hodnoty fieldu.

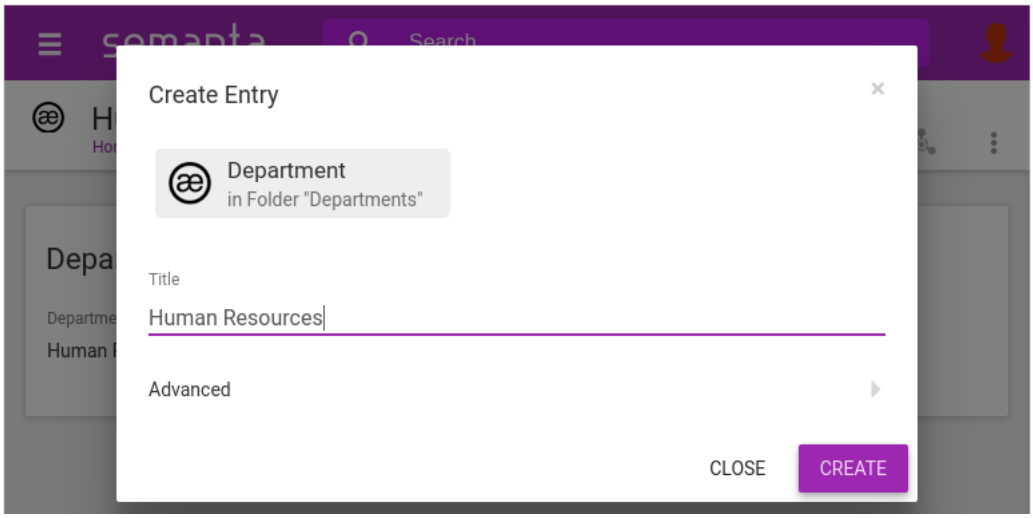
U oddělení chceme držet jeho název. Field s kódem `entryName` tedy bude reprezentovat název oddělení. Je typu `<plain>`, což je typ pro krátké textové pole. Kód `entryName` jsme nevybrali náhodně. Definice každé aplikace musí obsahovat field s tímto kódem, který se stane primárním názvem entry.

Vztah mezi oddělením a zaměstnancem musíme nadefinovat na aplikaci, ze kterého vazba vychází, v našem případě na aplikaci oddělení (Department) pomocí fieldu s kódem `hasEmployees`. Ten je typu `<relations>` a vyžaduje nadefinovat několik dalších vlastností.

Element `<generator-code>` použitý v rámci definice typu `<relations>` definuje tzv. *generator code* výraz, který použije platforma XF3 pro získání množiny entries, které mohou ve vztahu vystupovat. Výraz `${text}` bude nahrazen vstupem od uživatele, který napíše do pole našeptávače. Část před dvojtečkou určuje komponentu *(generátor)*, která dostane výraz ke zpracování a musí rozumět syntaxi části výrazu za dvojtečkou. Řekněme, že uživatel při editaci tohoto pole zadá do formulářového políčka text „Dolan“. Generator code výraz `xf3search:${text} (type:employee)` bude doplněn na `xf3search:Dolan (type:employee)`. Výraz bude zpracovávat generátor `xf3search`, který je napojen na vyhledávací služby platformy XF3. Pomocí těch se pokusí o nalezení entries aplikace *Employee*, obsahující text „Dolan“ jako hodnotu některého z jeho fieldů (aplikaci *Employee* nadefinujeme vzápětí).

Element `<relation-type>` definuje název vztahu. Název se zobrazuje při vizualizaci vztahů komponentou Nexus. Pojmenujeme vztah mezi oddělením a zaměstnancem jako `has-employee`.

Element `<display>` vybírá způsob zobrazení položek vztahu. Hodnotou `table` zvolíme tabulkový výpis, který podporuje stránkování a řazení.



Obrázek 2.2: Dialog pro vytvoření entry

Element `<multiple>` nastavuje kardinalitu vazby. Hodnota `true` způsobí, že v zdrojovém entry (zde Department) lze vytvořit vazbu na více entries. Z hlediska uživatele se to projeví tak, že může při editaci entry Department zvolit několik entries typu Employee.

Obrázky 2.2 a 2.3 ukazují, jak vypadají obrazovky pro vytvoření nového entry aplikace Department. V prvním kroku se zadává pouze název entry (field `entryName`), ve druhém kroku je možné vyplnit ostatní pole (fields). Zadali jsme „Human Resources“ jako název entry. Field typu `<relations>` při editaci zobrazuje dialog pro výběr s našeptáváním, ale zatím neexistují žádné entries typu Employee, které by bylo možné vybrat.

Definice aplikace obsahuje ještě element `<list>`, který definuje, jaké fieldy entry se mají zobrazovat v tabulkových výpisech entries tohoto typu. Lze zvolit jeden nebo více fieldů pomocí elementu `<fieldCode>`. V tabulkovém zobrazení aplikace Department se bude zobrazovat pouze field `entryName`.

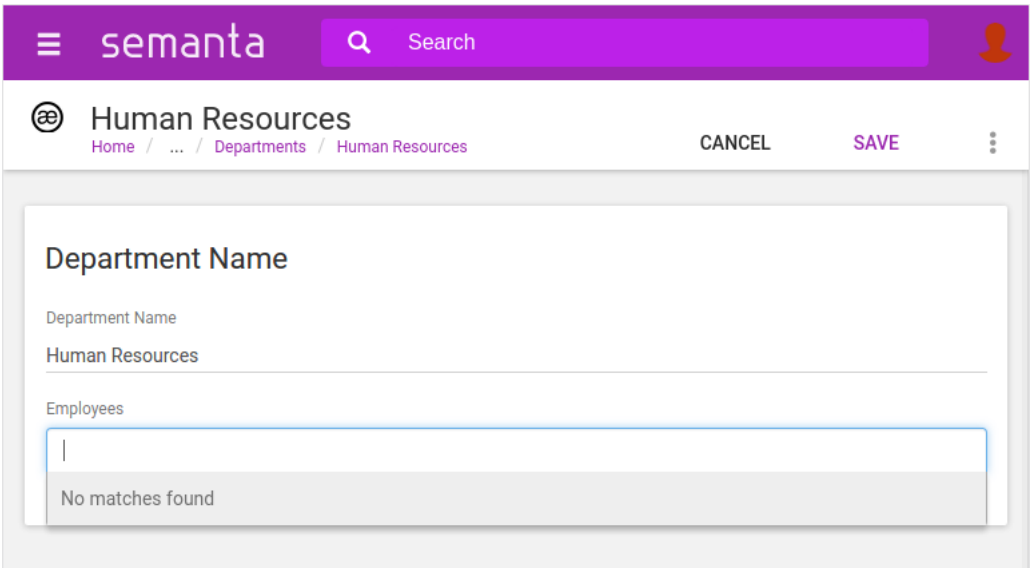
Poslední element `<listeners>` umožňuje pro události platformy zaregistrovat obslužné skripty. Ty budou vyvolány v případě, že se událost týká entry aplikace, ve které je obslužný skript v elementu `<listener>` zadefinován. Pro potřeby loaderu obslužné skripty nevyužijeme a nebudeme se jim dále věnovat.

2.2.2.2 Aplikace employee

Ukázka kódu 2.2 představuje definici aplikace `employee`, která definuje entry zaměstnance.

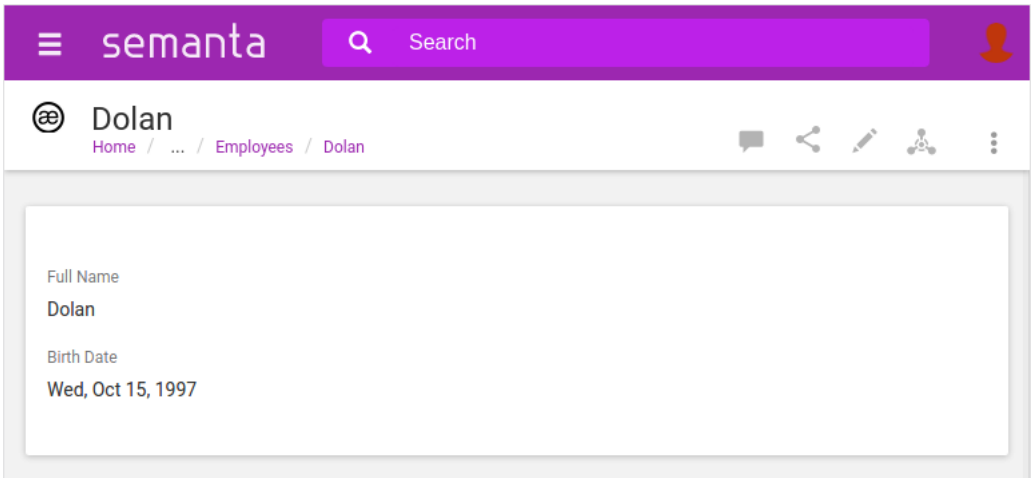
Fragment 2.2: Definice aplikace employee

```
<?xml version="1.0" encoding="UTF-8"?>
<xforms-apps xmlns="http://semanta.cz/schema/xf3">
    <xforms-app>
        <xid>employee</xid>
        <entry-name>Employee</entry-name>
```



Obrázek 2.3: Dialog pro editaci entry Department

```
        <structure>
            <row>
                <column>
                    <width>12</width>
                    <group>
                        <id>personal</id>
                        <fields>
                            <field>
                                <code>entryName</code>
                                <name>Full Name</name>
                                <type>
                                    <plain/>
                                </type>
                            </field>
                            <field>
                                <code>birthDate</code>
                                <name>Birth Date</name>
                                <type>
                                    <date-picker/>
                                </type>
                            </field>
                            <field>
                                <code>department</code>
                                <name>Department</name>
                                <type>
                                    <lucene-query>
                                        <query>hasEmployees:${entry.xid}</query>
                                        <display-as>table</display-as>
                                    </lucene-query>
                                </type>
                            </field>
                        </fields>
                    </group>
                </column>
            </row>
            <list>
                <fieldCode>entryName</fieldCode>
            </list>
        </structure>
        <listeners/>
    </xforms-app>
</xforms-apps>
```



Obrázek 2.4: Vytvořený objekt Employee

Oproti předchozí aplikaci je v aplikaci Employee přítomen field typu `<date-picker>` s kódem `birthDate`, kterým budeme reprezentovat datum narození zaměstnance.

Dále je zde field `department`, který realizuje viditelnost druhé strany vazby `has-employee`. Tato vazba je definována na aplikaci Department, na entries tohoto typu se vazba při editaci vytváří a jsou zde posléze odkazy na entry aplikace Employee. Na entry Employee chceme vazbu vidět z druhé strany, tj. mít zde k dispozici oddělení, do kterého zaměstnanec patří.

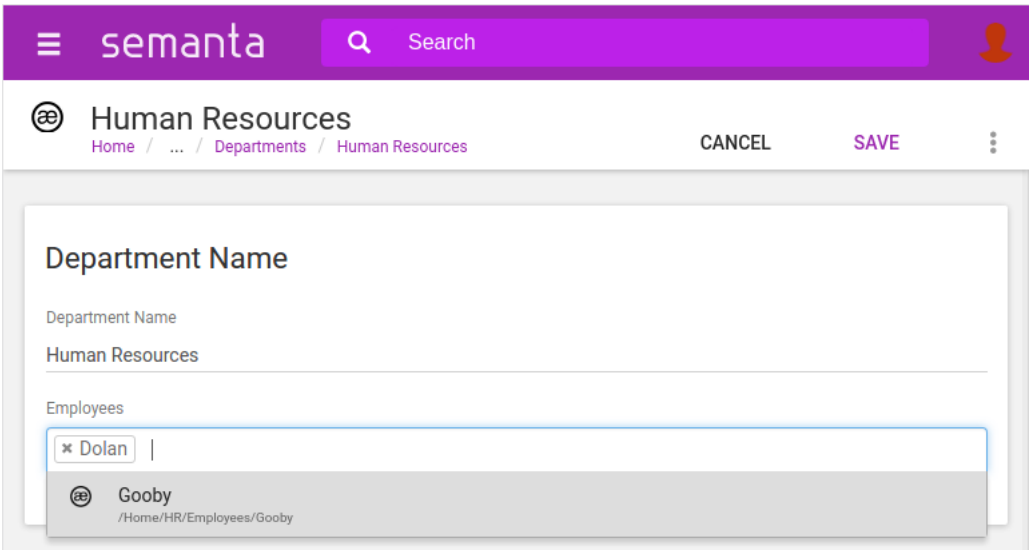
Opačnou stranu vazby zrealizujeme pomocí fieldu typu `<lucene-query>`. Ten stejně jako typ `<relations>` obsahuje element `<display-as>` pro výběr typu zobrazení výsledků. Podstatný je element `<query>`, kterým definujeme dotaz do služby vyhledávání platformy XF3, postavené na knihovně Apache Lucene.

Dotaz je v syntaxi knihovny Lucene, ale je navíc obohacený předzpracováním šablonovacím jazykem Velocity. XF3 využívá Velocity na mnoha místech své konfigurace a definuje sadu objektů, které jsou k dispozici. Velocity výraz `${entry.xid}` vede na atribut `xid` aktuálního entry.

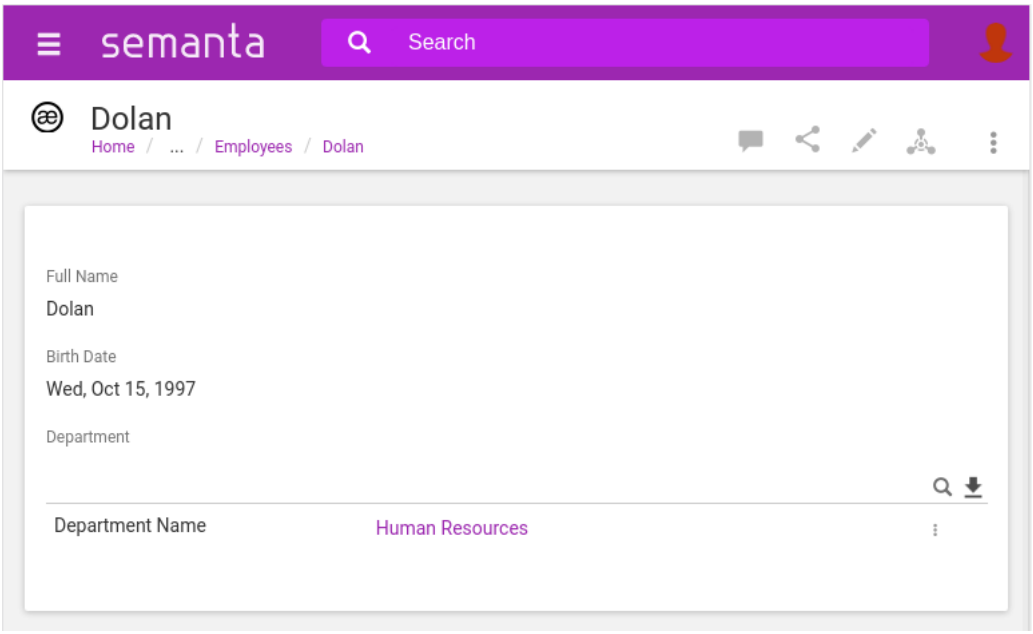
Vyhodnocením tohoto dotazu získáme všechny entries, jejichž hodnota fieldu `hasEmployees` referencuje xid aktuálního entry. Field s kódem `employees` jsme nadefinovali na aplikaci Department.

Vytvoříme dvě entry typu Employee. Obrázek 2.4 ukazuje, jak vypadá výsledné entry.

Nyní vytvoříme na entry Human Resources typu Department vazbu na obě entries Employee. Obrázek 2.5 zobrazuje, jak vypadá editace vazby na entry, ze kterého vazba vychází. Obrázek 2.6 ukazuje vazbu, jak vypadá z pohledu cílového entry (Employee). Na cílovém entry se automaticky objevila informace o oddělení (Department).



Obrázek 2.5: Vytvoření vazby has-employee na vlastnícím entry



Obrázek 2.6: Vazba has-employee z pohledu cílového entry

2.2.3 Definice nahrávací úlohy

Nahrání dat z externího zdroje se skládá ze dvou částí. První část je položení dotazu do relační databáze a druhá část řeší mapování řádku výsledku dotazu na fieldy konkrétní aplikace XF3.

V rámci definice nahrávací úlohy lze použít šablonovací jazyk Velocity. Jazyk Velocity umožňuje referencovat objekty, které jsou nadefinovány v tzv. *kontextu*. V rámci platformy XF3 existuje standardní XF3 Velocity kontext s objekty, které reprezentují jednotlivé služby platformy. Z pohledu nahrávací úlohy je z XF3 Velocity kontextu důležitý pouze objekt `LoadingUtils`, který obsahuje funkcionalitu pro podporu nahrávacích úloh.

Podstatný je fakt, že kontext pro Velocity v rámci definice nahrávací úlohy postupně obohacován o výsledky dotazů. V rámci definice SQL dotazu je k dispozici pouze standardní Velocity XF3 kontext, zatímco v rámci zpracování výsledků dotazu už kontext obsahuje hodnoty jednotlivých sloupců výsledku dotazu pod proměnnými, jejichž názvy odpovídají názvům sloupců.

Následuje definice nahrávací úlohy pro vytvoření entries typu Department a Employees z dat v relační databázi. Fragment 2.3 naznačuje celkovou strukturu nahrávací úlohy. Definice obsahuje kořenový element `<loading-instructions>`, který obsahuje jeden element `<job-id>`, definující název jobu. Název se zobrazuje v uživatelském rozhraní pro spouštění jobů, viz 2.7.

Fragment 2.3: Struktura definice nahrávací úlohy

```
<?xml version="1.0" encoding="UTF-8"?>
<loading-instructions xmlns="http://semanta.cz/schema/xf3/load">
    <loading-instruction>
        <job-id>load-mysql-sample</job-id>

        <for-each-row>
            <sub-job-id>employees</sub-job-id>
            ...
        </for-each-row>

        <for-each-row>
            <sub-job-id>departments</sub-job-id>
            ...
        </for-each-row>

    </loading-instruction>
</loading-instructions>
```

Element `<loading-instructions>` obsahuje jeden až N elementů `<for-each-row>`, přičemž každý z nich definuje jednu *podúlohu*, který je základní jednotkou nahrávací úlohy. Podúloha definuje SQL dotaz spolu s transformací jeho výsledků na entries jedné aplikace. V našem případě máme dva elementy `<for-each-row>`. První definuje podúlohu pro vytvoření entries aplikace Employee, druhý podúlohu pro aplikaci Department. Podíváme se podrobněji na podúlohu pro aplikaci Department, která je zajímavá tím, že v jeho rámci vytváříme vazby has-employee.

Fragment 2.4 ukazuje definici podúlohy pro typ Department. Každá podúloha má identifikátor, definovaný elementem `<sub-job-id>`. Element `<of-query>` definuje SQL dotaz, který má být položen proti relační databázi. Výsledek

SQL dotazu je zpracován pomocí mapování, definovaném v rámci elementu `<define-entry>`. Mapování jednak určuje cílovou aplikaci pro vznikající entry a dále mapování výsledků dotazu na fieldy této aplikace. Poslední volitelnou částí definice podúlohy je sekundární SQL dotaz (element `<process-all-rows>`), který je proveden pro každý řádek výsledku primárního dotazu `<of-query>`. Pomocí sekundárního dotazu lze například pro vznikající entry typu Department položit SQL dotaz do vazební tabulky a na základě výsledku tohoto dotazu vytvořit relace mezi entry Department a Employee.

2.2.3.1 Primární SQL dotaz

Element `<of-query>` obsahuje podelement pro definici dotazu. V případě loadu z relační databáze je to element `<sql>`.

Ten obsahuje element `<connection-id>`, který odkazuje na připojení do relační databáze, nadefinované na úrovni globální konfigurace platformy. Ta z pohledu této práce není podstatná.

Samotný SQL dotaz je definován v elementu `<query-body>`. V našem případě definujeme jednoduchý dotaz na dva sloupce z tabulky `departments`.

2.2.3.2 Mapování výsledků dotazu

Element `<define-entry>` definuje mapování výsledku SQL dotazu. Pro každý řádek výsledku bude vytvořeno jedno entry.

Element `<app>` určuje aplikaci, jejíž entry bude vytvořeno (očekává se xid aplikace).

Element `<parent>` definuje xid entry, které bude rodičem vznikajícího entry. Entry lze hierarchicky vnořovat, pokud to definice jejich aplikace umožňují. V našem případě budou všechny entry typu Department umístěny pod jedno rodičovské entry.

Abychom mohli provést vyhledání rodiče bez závislosti na konkrétním obsahu, použijeme obrat, který vyhledá entry podle speciálního fieldu – podle integračního kódu. U tohoto fieldu předpokládáme, že jeho hodnota je unikátní přes všechny entry v XF3. Metoda pro vyhledání entry se volá (využitím syntaxe jazyka Velocity) přes proměnnou `lu`, která vede na objekt třídy `LoadingUtils`.

Element `<parent-field>` určuje kód fieldu na výše zmíněném rodičovském entry, které může držet vznikající entry. Využíváme aplikaci `folder` (adresář) z výchozí sady aplikací dodávané s platformou XF3. O té víme, že obsahuje pole s kódem `entries` pro držení potomků.

Element `<xid>` definuje identifikátor (xid) nově vznikajícího entry. V našem SQL dotazu je jeden ze sloupců výsledku primární klíč – `dept_no`. V kontextu pro Velocity jsou v rámci nahrávací úlohy přístupné hodnoty jednotlivých sloupců aktuálního řádku vyhodnoceného dotazu. Tyto hodnoty jsou k dispozici pod proměnnými, pojmenovanými podle názvů sloupců ve výsledku

SQL dotazu. V našem případě hodnotu sloupce s primárním klíčem máme k dispozici pod proměnnou `$dept_name` a použijeme ji jako xid.

Element `<name>` definuje název entry. Název entry odpovídá povinnému fieldu `entryName`, ale definuje se zvlášť od ostatních fieldů. Použijeme pro název hodnotu sloupce `$dept_name`.

Element `<fields>` slouží k mapování hodnot výsledku na fieldy aplikace. V případě aplikace Department máme kromě povinného fieldu `entryName` ještě relační field `hasEmployee`. Pro vytvoření jeho hodnotu ale potřebujeme položit dotaz do vazební tabulky, proto jí nejsme schopni na tomto místě naplnit. Použití elementu `<fields>` si ještě ukážeme na příkladu nahrávací úlohy pro Employee.

2.2.3.3 Sekundární SQL dotaz

Element `<process-all-rows>` definuje sekundární dotaz, který bude položen jednou pro každý řádek výsledku primárního dotazu. Typicky slouží k dohledání relací pro vznikající entry.

Element `<of-query>` má stejnou strukturu, jako v případě primárního SQL dotazu. Definuje id připojení do databáze a samotný SQL dotaz. Opět lze využít šablonovací jazyk Velocity. Oproti primárnímu SQL dotazu je ve Velocity kontextu u definice sekundárního dotazu k dispozici aktuálně zpracovávaný řádek výsledku primárního dotazu.

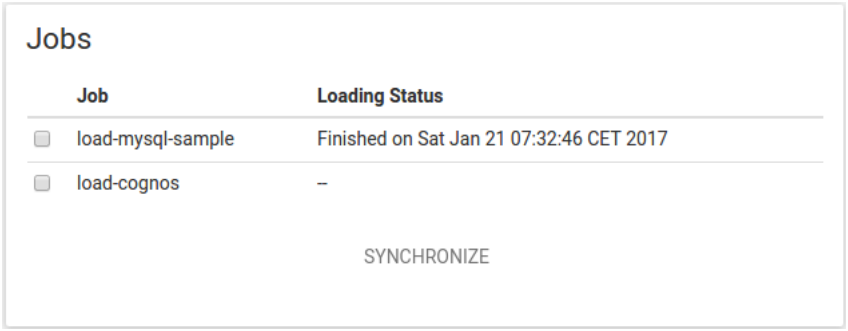
Tohoto využíváme v definici podúlohy 2.4. V klauzuli `WHERE` požadujeme, aby se hodnota sloupce výsledku `dept_no` rovnala hodnotě proměnné kontextu `$dept_no`. Tato proměnná obsahuje hodnotu sloupce `dept_no` z výsledku primárního SQL dotazu. Tímto položíme do vazební tabulky `dept_emp` dotaz na řádky, týkající se pouze právě zpracovávaného oddělení.

Element `<set-fields>` má stejnou funkci, jako výše zmíněný element `<fields>`. Umožňuje naplnit fieldy entry na základě výsledku sekundárního SQL dotazu. V našem případě takto naplníme field `hasEmployees`, který definuje vazbu mezi aplikacemi Department a Employee. Hodnoty vazby jsou interně reprezentovány speciálním objektem třídy `Targets`, který musíme v rámci definice mapování vytvořit. Objekt `Targets` obsahuje seznam identifikátorů (xidů), které jsou cílové entries vazby.

Využijeme k tomu opět metodu služby `LoadingUtils`. Metodě `targetsFromRows` předáme celý výsledek sekundárního SQL dotazu, dostupný pod proměnnou `$allRows`. Druhým parametrem je název sloupce z výsledku `$allRows`, který obsahuje hodnoty identifikátoru (xid) entries, na které má mířit zakládaná vazba. Tyto hodnoty xid se stanou obsahem výše zmíněného objektu `Targets` a tento se stane hodnotou fieldu (v našem případě fieldu `hasEmployees`).

Fragment 2.4: Definice loadu pro entry Department

```
<for-each-row>
    <sub-job-id>departments</sub-job-id>
    <of-query>
        <sql>
```

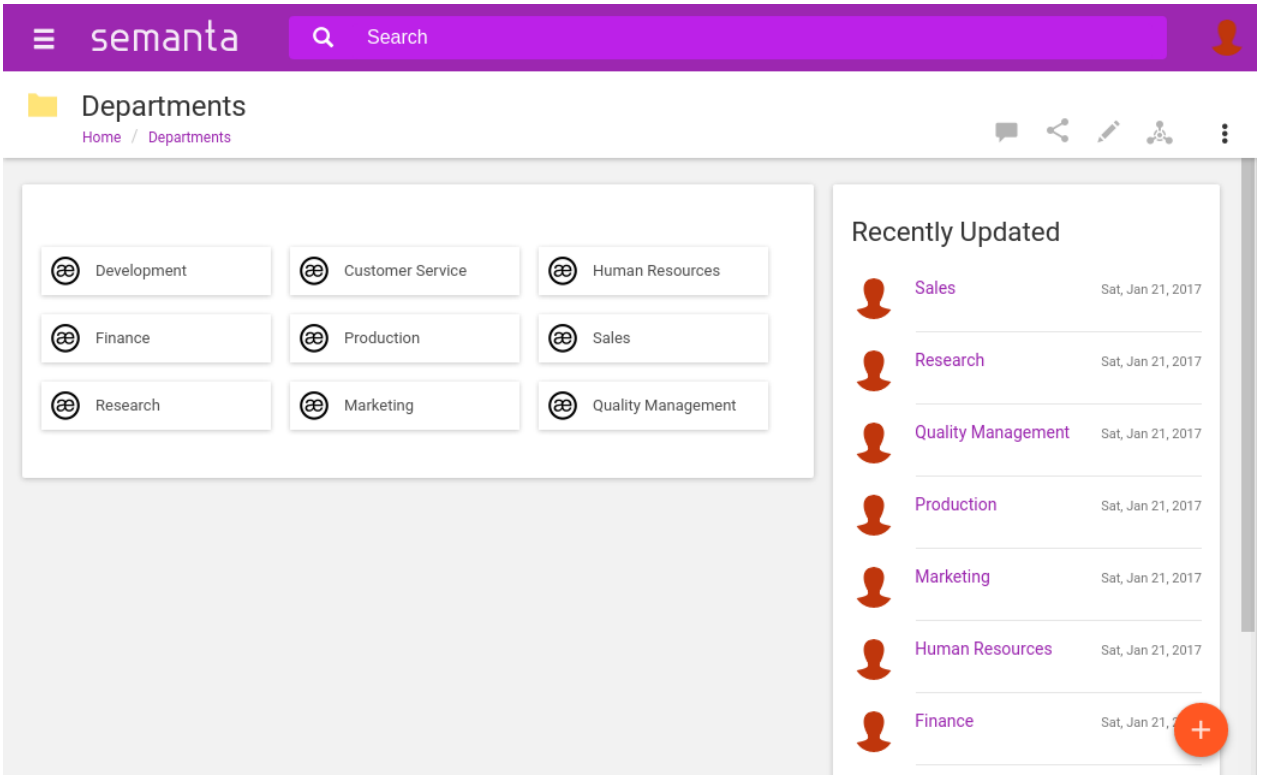


Obrázek 2.7: Uživatelské rozhraní pro obsluhu nahrávacích úloh

```
            <connection-id>mysql-sample</connection-id>
            <query-body>
                SELECT
                    dept_no,
                    dept_name
                FROM
                    departments
            </query-body>
        </sql>
    </of-query>
    <define-entry>
        <app>department</app>
        <parent>$lu.getEntryForIntegrationCode("sample-departments").getXid()</parent>
        <parent-field>entries</parent-field>
        <xid>$dept_no</xid>
        <name>$dept_name</name>
        <fields></fields>
        <process-all-rows>
            <of-query>
                <sql>
                    <connection-id>mysql-sample</connection-id>
                    <query-body>
                        SELECT
                            emp_no, dept_no
                        FROM
                            dept_emp
                        WHERE
                            dept_no = "$dept_no"
                    </query-body>
                </sql>
            </of-query>
            <set-fields>
                #set($hasEmployees=$lu.targetsFromRows($allRows, "emp_no"))
            </set-fields>
        </process-all-rows>
    </define-entry>
</for-each-row>
```

Obrázek 2.7 ukazuje uživatelské rozhraní pro ovládání nahrávacích úloh. Vidíme stav po úspěšném provedení úlohy.

Obrázek 2.8 zachycuje entry typu Departments vzniklé nahráním. Detail jednoho entry zachycuje obrázek 2.9. Jedná se oddělení *Finance*. Vidíme, že toto oddělení obsahuje 16 154 zaměstnanců (entries aplikace Employee).

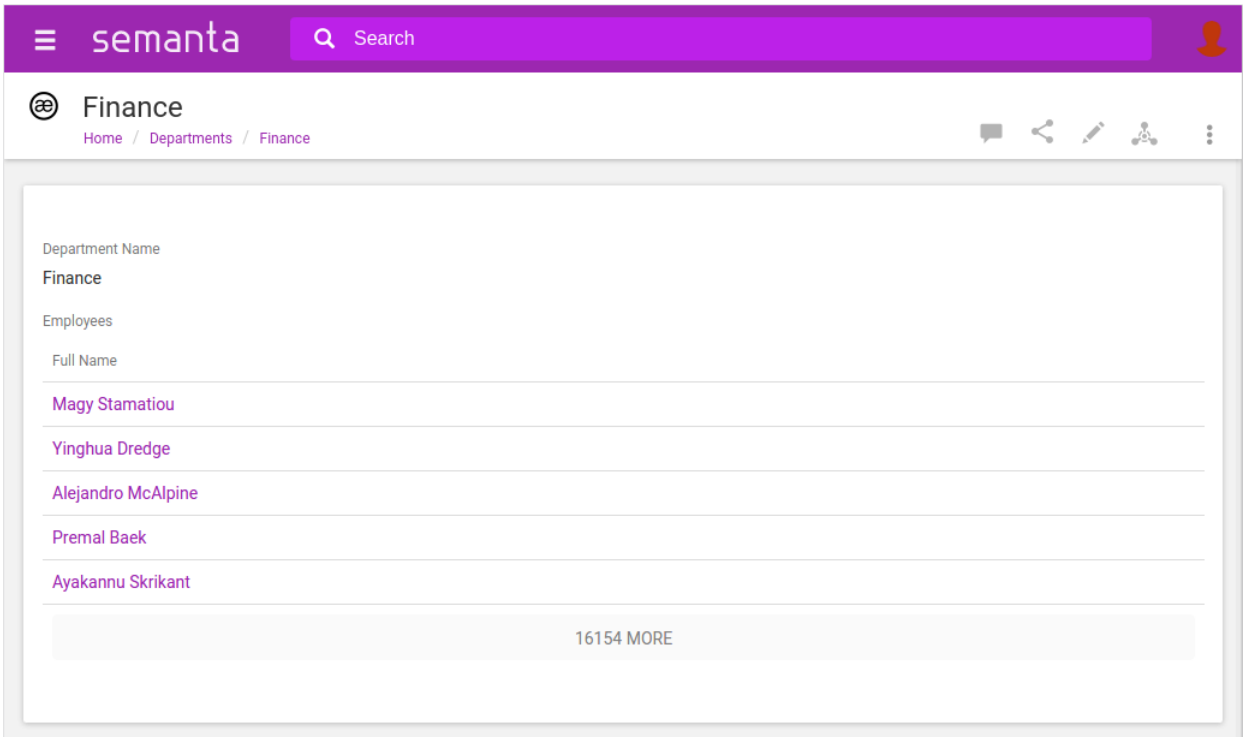


Obrázek 2.8: Entry vzniklé nahráním

2.3 Shrnutí

Na jednoduchém příkladě jsme si demonstrovali vlastnosti platformy XF3, které jsou klíčové z pohledu katalogu reportů. Tím jsme zároveň ověřili, že XF3 je vhodnou technologií pro implementaci katalogu. Platforma umožňuje deklarativně definovat aplikace pro jednotlivé typy objektů platformy IBM Cognos. Nadefinováním nahrávací úlohy získáme schopnost nahrát z relační databáze do jednotlivých aplikací data získaná z IBM Cognos.

V prvních dvou kapitolách jsme se věnovali zkoumání zdrojové platformy IBM Cognos a platformě pro realizaci prezentace dat. V dalších kapitolách se zaměříme na implementační část práce.



Obrázek 2.9: Detail entry vzniklého nahráním

Kapitola 3

Realizace loaderu

Tato kapitola se zabývá realizací na straně zdroje dat – získáním dat z platformy IBM Cognos. Tuto část jsme v úvodu pojmenovali jako *loader*. V rámci této kapitoly budeme postupně rozpracovávat představu o činnosti loaderu – zejména části, týkající se získání a zpracování dat z platformy Cognos.

Úloha loaderu spočívá v připojení se k běžící instanci IBM Cognos, získání potřebných dat pomocí API platformy, jejich zpracování a nakonec zápisu zpracovaných dat do relační databáze. Odtud si data převezme aplikace report katalogu pomocí mechanismu nahrávací úlohy platformy XF3, který jsme popsali v kapitole 2. Implementaci tohoto mechanismu popíšeme v kapitole 4.

Součástí úlohy je také načtení vstupních dat, které určují parametry dalšího chodu konektoru. Mezi těmito vstupními daty je seznam reportů ke zpracování, přihlašovací údaje do platformy IBM Cognos a parametry databáze, do které loader provede zápis svého výstupu.

3.1 Získání dat z API IBM Cognos

V kapitole 1 bylo zmíněno, že objekty platformy IBM Cognos se získávají pomocí služby Content Manager Service. Na objekty se dotazuje pomocí search path výrazů. Konektor na vstupu dostane search path výraz, který specifikuje reporty ke zpracování.

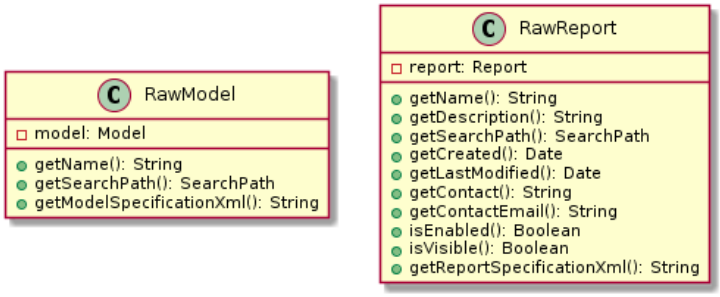
Pomocí metody `query` služby Content Manager Service získáme instance třídy `Report`, kterou definuje API IBM Cognos. Objekt `Report` poskytuje mimo jiné search path pro datový model (reprezentovaný objektem `Model`), ze kterého čerpá data. Dotazem na Content Manager Service získame na základě search path příslušný objekt `Model`.

Jsme tedy schopni na základě cestě k reportu získat z API objekt reportu a objekt jeho datového modelu. Tyto objekty API nám přímo poskytnou některé informace, které o reportech, respektive modelech, chceme katalogizovat. Atributy objektů API jsou ale pouze jedním ze zdrojů informací, které potřebujeme získat.

Další část informací je obsažena v interní specifikaci reportu a modelu, která je v XML formátu definovaným platformou Cognos. Přístup k těmto XML souborům je umožněn pomocí atributů technického rázu, které třídy API `Report` a `Model` definují. Zpracování těchto XML dokumentů musíme implementovat v rámci loaderu vlastními prostředky, Cognos SDK k tomu neposkytuje žádnou podporu.

Třetí část informací se nachází v tzv. *data lineage* dokumentech. Data lineage vyjadřuje tok dat mezi objekty – z jakých jiných objektů získávají objekty data. Zajímá nás jednak lineage datových položek reportu (závislost datových položek reportu na objektech datového modelu) a dále lineage objektů datového modelu (závislost objektů datového modelu mezi sebou).

V kapitole 1 jsme se dozvěděli, že informace o lineage poskytuje služba Metadata Service. Služba přijímá dotaz na lineage v podobě XML dokumentu



Obrázek 3.1: Doménové objekty reprezentující nezpracovaný report a model

a vrací specifikaci lineage v podobě jiného XML dokumentu.

Abychom mohli položit dotaz na lineage datových položek reportu, potřebujeme nejprve všechny tyto datové položky znát. Informace o datových položkách reportu je součástí struktury reportu, kterou získáme zpracováním XML specifikace reportu.

Pro položení dotazu na lineage objektů balíčku datového modelu je nutné získat seznam identifikátorů *(objectId)* objektů modelu zpracováním XML specifikace modelu.

Kombinací informací získaných z 1. atributů objektů API, 2. interní XML reprezentace objektů platformy a 3. specifikace data lineage ve formátu XML získáme celou sadu informací, kterou o objektech platformy IBM Cognos chceme prezentovat.

Postup získání těchto tří zdrojů informací není náhodný. Nejprve je nutné získat objekty API. Z objektů API lze získat interní XML specifikace, jejichž zpracování nám následně umožní získat dokumenty o data lineage.

Ve zbytku kapitoly budeme zde nastíněný postup rozpracovávat do většího detailu.

3.2 Tři úrovně abstrakcí pro objekty reportu a modelu

V loaderu zavádíme vlastní doménové objekty pro objekt reportu a modelu, s kterými budou pracovat jednotlivé služby, ze kterých se loader skládá. Zavádíme tři typy doménových objektů.

3.2.1 Abstrakce zapouzdřující objekty API

Prvním typem jsou objekty, reprezentující nezpracovaný report a model. Jsou to objekty `RawReport` a `RawModel`, které vidíme na diagramu 3.1. Tyto objekty zapouzdřují objekty API Cognos (`Report` a `Model`) a zpřístupňují pouze ty jejich vlastnosti, které jsou z hlediska loaderu relevantní.

Objekty API zapouzdříme hned v momentě jejich získání a dále pracujeme pouze s námi definovanými abstrakcemi `RawReport` a `RawModel`. Tímto se snažíme omezit výskyt objektů z API v rámci vlastního kódu konektoru na minimum. Preferujeme, aby v návrhu konektoru byla přítomna pouze námi definovaná rozhraní namísto objektů API IBM Cognos. Kontrakt objektů API je velmi nepřehledný. Objekty mají velké množství metod a *Javadoc* dokumentace k API není distribuována v podobě, umožňující její využití v *IDE*.

Podobným způsobem zapouzdřujeme i objekty API, které reprezentují jednotlivé služby (například objekt API `ContentManagerService_PortType` pro Content Manager Service). Abstrakcím zapouzdřujícím objekty služeb API se budeme věnovat v rámci popisu jednotlivých podúloh.

V terminologii *Design Patterns* jsou naše zapouzdřující objekty realizací návrhového vzoru *Facade*.

3.2.2 Abstrakce pro výsledky podúloh zpracování

Ustanovili jsme, že zpracování modelu a reportu se skládá ze tří podúloh. Pro výsledky jednotlivých podúloh zavádíme objekty, které reprezentují výsledek konkrétní podúlohy. Například pro podúlohu zpracování XML specifikace reportu (jejíž výsledkem je struktura reportu) zavádíme třídu `ReportStructure`.

Abstrakce pro výsledky jednotlivých podúloh zavedeme v rámci popisu návrhu služeb loaderu zodpovědných za zpracování těchto podúloh.

3.2.3 Abstrakce pro zpracovaný report a model

Kompletně zpracovaný datový model reprezentuje třída `ProcessedModel`, pro report zavádíme obdobnou třídu `ProcessedReport`. Tyto objekty se skládají z řady dalších objektů. Například report se skládá ze stránek, které jsou reprezentovány instancemi třídy `ReportPage`. Obdobně model se skládá z balíčků, které reprezentujeme instancemi třídy `ProcessedPackage`.

Objektový model pro zpracovaný report a model podrobněji popíšeme v rámci popisu úloh zpracování modelu a reportu.

3.3 Barevné schéma v sekvenčních diagramech

Pro snazší orientaci v *sekvenčních* diagramech přiřazujeme jednotlivým typům objektů odlišné barvy.

Objekty definované Cognos API jsou vyobrazeny oranžovou barvou.

Objekty zapouzdřující objekty Cognos API jsou vyobrazeny červenou barvou.

Objekty vyžadující další transformace, aby se mohly stát součástí výsledku (například objekt struktury reportu), jsou vyobrazeny krémovou barvou.

Kompletně zpracované objekty, které jsou součástí výsledku a nevyžadující další transformace, jsou vyobrazeny zelenou barvou.

Služby loaderu jsou vyobrazeny modrou barvou.

3.4 Získání objektů reportu a modelu

V první fázi činnosti loaderu je nutné získat objekty API `Report` a `Model`. Výsledkem této fáze jsou objekty `RawReport` a `RawModel`, které objekty API zapouzdřují (viz diagram 3.1).

Sekvenční diagram 3.2 zobrazuje proces pro získání objektů API.

Zavádíme rozhraní `ReportFetcher`, které je zodpovědné za získání jednoho objektu `Report` na základě search path, která je vstupním parametrem spuštění loaderu. Obdobně zavedeme rozhraní `ModelFetcher` pro získání objektu `Model` na základě search path, kterou získáme z objektu `Report`.

Protože více reportů typicky sdílí stejný datový model, chceme předejít v případě zpracování více reportů naráz opakovanému dotazování se na stejný model. Pro tento účel zavedeme rozhraní `ReportAccumulator`, které dokáže pro sadu reportů sdělit množinu modelů, na které se reporty odkazují.

Získávání objektů API vyžaduje předem specifikovat, které atributy objektů mají být naplněny daty. V opačném případě jsou atributy prázdné. Blíže viz sekce 1.4.1.

3.5 Přehled zpracování modelu

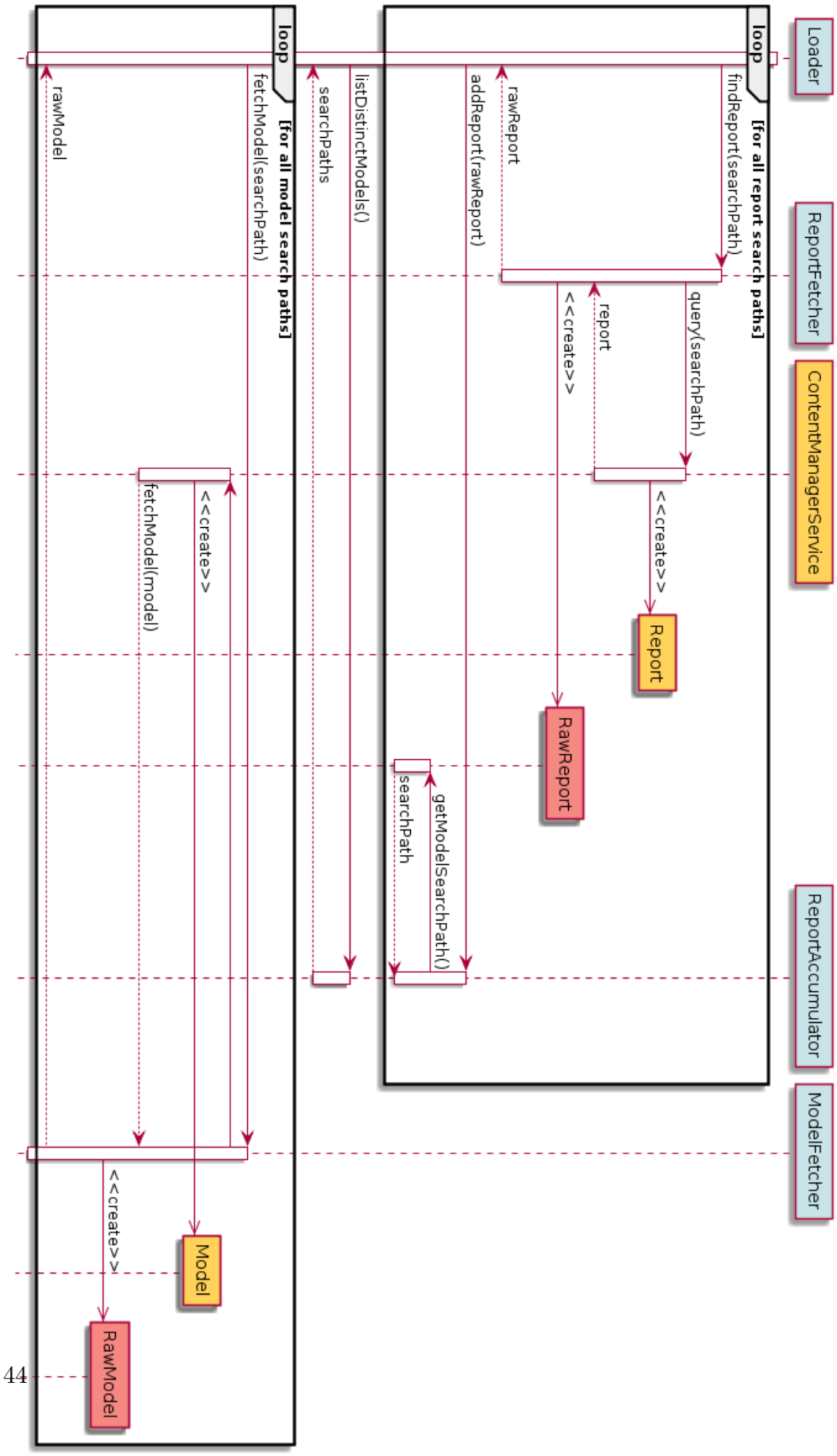
Za zpracování modelu (reprezentovaného objektem `RawModel`) je zodpovědná služba konektoru `ModelProcessor`. Ta vrací na výstupu objekt zpracovaného modelu `ProcessedModel`.

Přehled celého zpracování objektu modelu je naznačen na diagramu 3.3. Zpracování modelu se skládá ze dvou podúloh.

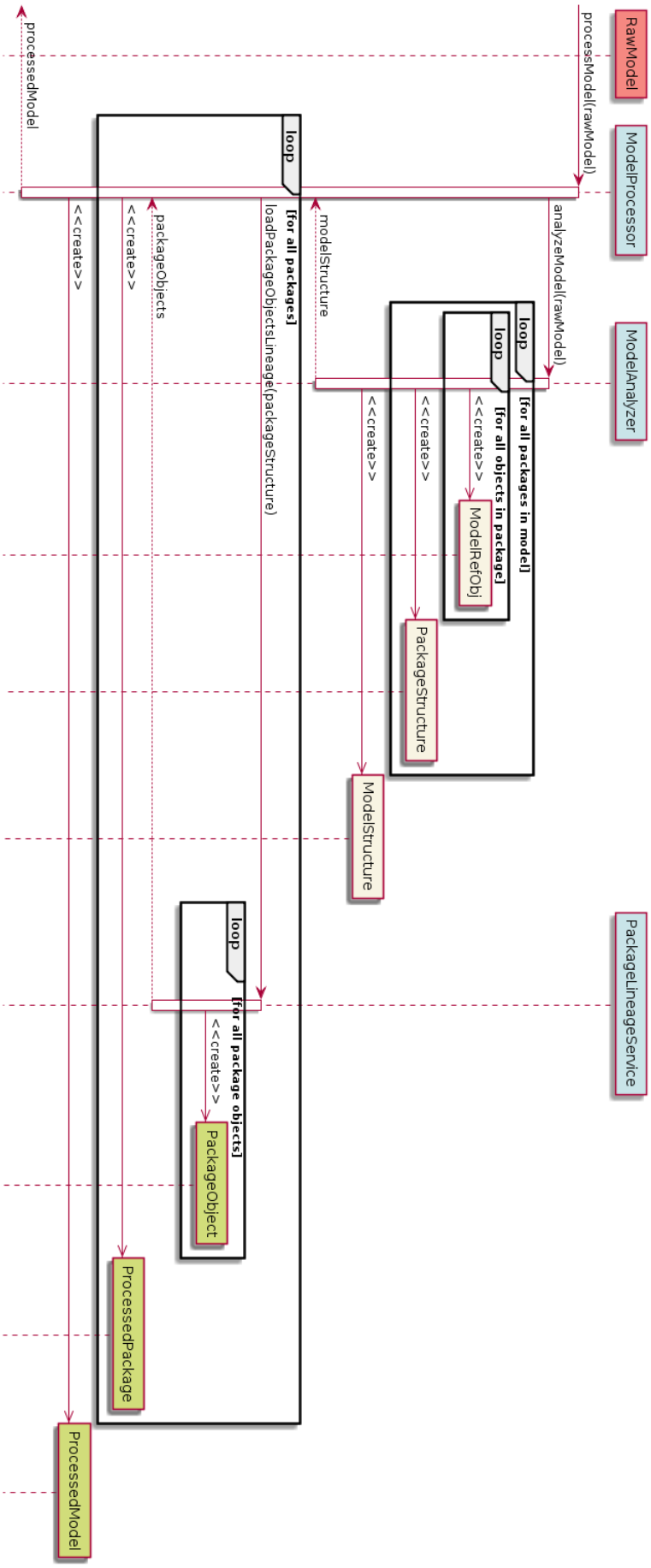
První podúlohou je zpracování interní XML specifikace modelu. Za to je zodpovědná služba loaderu `ModelAnalyzer`. Výsledkem této podúlohy je částečně zpracovaný model, reprezentovaný objektovou struktura z diagramu 3.4. Třída `ModelStructure` pro částečně zpracovaný model představuje strukturu modelu, získanou z XML specifikace. Struktura modelu se dále skládá z balíčků modelu.

Balíček je reprezentován třídou `PackageStructure`. Informace o balíčku modelu je v tuto chvíli omezena na seznam identifikátorů objektů v balíčku. Identifikátor objektu modelu dále nazýváme referencí na objekt modelu a je reprezentován třídou `ModelRefObj`.

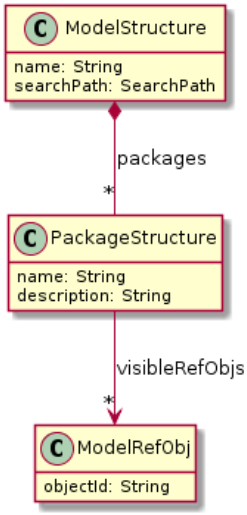
Druhou podúlohou je získání a zpracování data lineage dokumentu o objektech modelu. Vstupem pro tuto úlohu je seznam referencí na objekty modelu. Seznam referencí na objekty modelu získáme z objektu `PackageStructure`, který je součástí výstupu minulé podúlohy. Za získání a zpracování data lineage o objektech modelu je zodpovědná služba loaderu `PackageLineageService`.



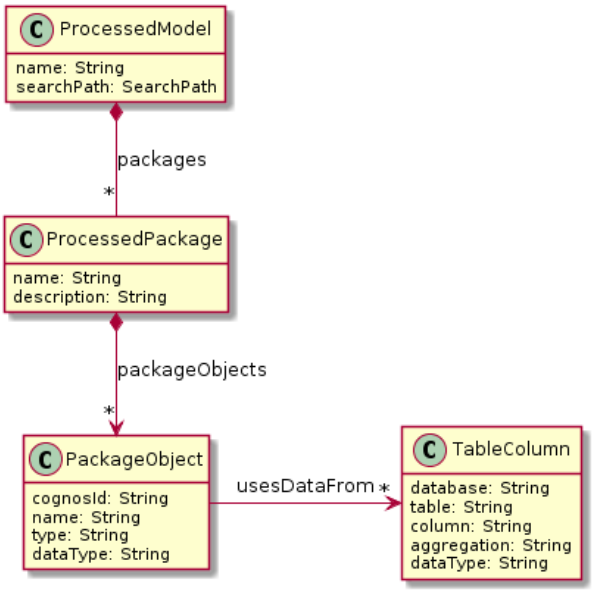
Obrázek 3.2: Sekvenční diagram získání objektů Report a Model z API



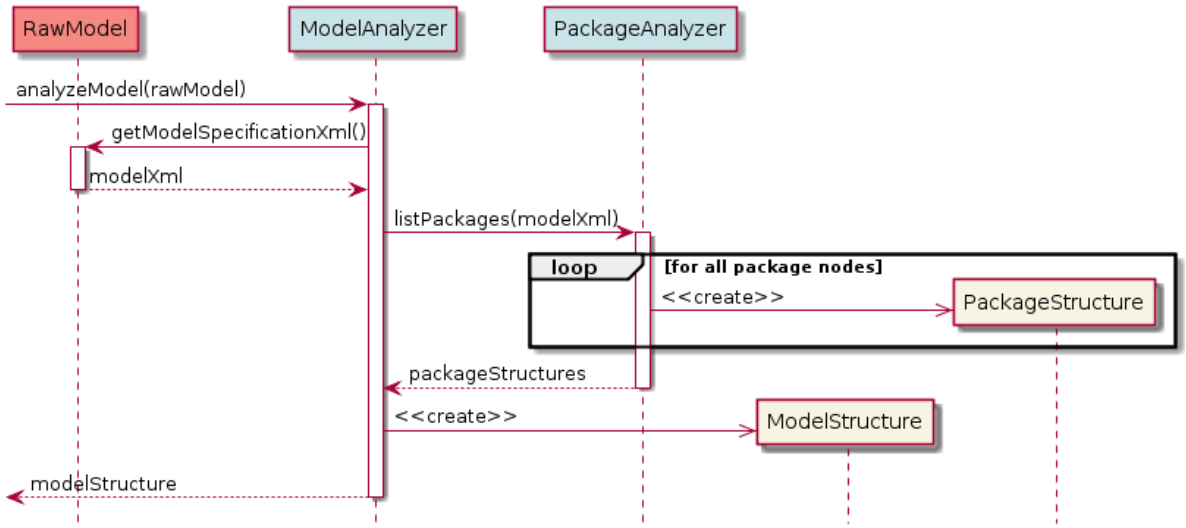
Obrázek 3.3: Přehled zpracování objektu Model z API



Obrázek 3.4: Objekty reprezentující částečně zpracovaný model



Obrázek 3.5: Objekty zpracovaného modelu



Obrázek 3.6: Diagram zpracování struktury modelu

Výsledkem celého zpracování modelu je objektová struktura, kterou zobrazuje diagram 3.5. Třída `ProcessedModel` reprezentuje kompletně zpracovaný datový model, třída `ProcessedPackage` balíček modelu, který obsahuje objekt datového modelu (třída `PackageObject`). Třída `TableColumn` představuje informaci o relačních datových zdrojích, které jsme schopni dohledat z data lineage pro některé objekty datového modelu.

3.6 Zpracování XML specifikace modelu

Tuto podúlohu realizuje služba `ModelAnalyzer` způsobem, který je naznačen diagramem 3.6.

Služba `ModelAnalyzer` získá z objektu nezpracovaného modelu `RawModel` interní XML specifikaci modelu voláním metody `getModelSpecificationXml()`. Na základě analýzy XML dokumentu vytvoří objekty, reprezentující částečně zpracované balíčky modelu (`PacakgeStructure`). Ty obsahují seznam referencí na objekty modelu (`ModelRefObj`).

Získání informace o balíčcích je delegováno na samostatnou službu `PackageAnalyzer`. Implementace služeb `ModelAnalyzer` a `PackageAnalyzer` využívá pro vyhledání podstromů XML dokumentu a jejich následné zpracování jazyk *XPath*.

3.6.1 Jazyk XPath

XPath je dotazovací jazyk pro výběr uzlů z XML dokumentu. Práci s XPath na platformě JVM umožňuje knihovna `java.xml`. Práce s XPath abstrakcemi poskytovanými touto knihovnou je poměrně nepříjemná (knihovna například nepodporuje iterátory, což znemožňuje použít běžné obraty jazyka Java). Proto

v rámci této práce vznikla jednoduchá knihovna *XPath Selectors* pro práci s XPath.

3.6.2 Vlastní knihovna XPath Selectors

Výhodou XPath Selectors je možnost tvořit XPath dotaz postupně skládání selektorů. Ukázka 3.1 a demonstruje použití knihovny *JUnit* testem. Ukázka využívá vzorový XML soubor [4] (3.2). Knihovna je inspirována komponentou *Selectors* [5] z knihovny *Scrapy* pro jazyk Python.

Fragment 3.1: Ukázka použití knihovny XPath Selectors

```
XPathSelector document = new DocumentXPathSelector(booksXml);
XPathSelector catalog = document.query("/catalog");
assertThat(catalog.resultSize(), is(1));
XPathSelector book101Genre = catalog.query("book[@id='bk101']/genre");
assertThat(book101Genre.resultSize(), is(1));
String book1GenreText = book101Genre.query("text()").singleResult().getNodeValue();
assertThat(book1GenreText, is(equalTo("Computer")));
```

Fragment 3.2: XML dokument využitý v ukázce knihovny

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk108">
      <author>Knorr, Stefan</author>
      <title>Creepy Crawlies</title>
      <genre>Horror</genre>
      <price>4.95</price>
      <publish_date>2000-12-06</publish_date>
      <description>An anthology of horror stories about roaches,
      centipedes, scorpions and other insects.</description>
   </book>
</catalog>
```

Z pohledu vnitřního uspořádání konektoru se knihovna pro XPath Selectors objevuje pouze v implementacích jednotlivých služeb a architekturu loaderu nijak neovlivňuje.

Knihovna XPath Selectors umožnuje snadné vyhledávání podstromů v XML dokumentu, ale následné parsování těchto podstromů stále obnáší mnoho repetitivní práce. Proto využíváme knihovnu *Simple XML* pro automatické vytváření Java objektů z těchto podstromů.

3.6.3 Knihovna Simple XML

Knihovna Simple XML umožňuje automatické vytváření objektů z uzlů XML dokumentu na základě anotací. Na podobném principu pracuje standardní knihovna *JAXB*, která je součástí specifikace *Java EE*. Nevýhodou technologie JAXB je, že vyžaduje XSD schémata pro generování tříd objektů. V našem případě sice máme XSD schémata interních XML dokumentů platformy Cognos, protože jsou dodávány v rámci *Cognos SDK*. Zajímají nás ale pouze malé podmnožiny celých schémat, což není scénář, pro který by byla knihovna JAXB navržena.

3.6.4 Získání informací o balíčcích modelu z XML specifikace

Úlohou abstrakce `PackageAnalyzer` je získat z XML specifikace modelu informaci o balíčcích, které model definuje. Fragmenty 3.3 a 3.4 ukazují části XML dokumentu, které v rámci této úlohy potřebujeme zpracovat.

Balíček datového modelu je definován elementem `<package>`. Ten obsahuje metadata o balíčku a také název interní specifikaci balíčku, která je v rámci XML modelu vyjádřena elementem `<securityView>`. Element `<securityView>` obsahuje sadu referencí – identifikátorů *objectId* na jednotlivé objekty datového modelu. Odkazy na objekty modelu jsou v XML vyjádřeny elementy `<refobj>`. Jak víme, reference na objekty modelu reprezentujeme doménovým objektem `ModelRefObj` (viz diagram 3.4).

Pro implementaci třídy `XPathPackageAnalyzer` využíváme kombinaci knihovny XPath Selectors s knihovnou Simple XML. Knihovna XPath Selectors je použita k získání uzlu v XML modelu, odpovídající elementům `packages` a `securityViews`. Objekty selektorů nám umožní výsledek transformovat na řetězce, které posléze parsujeme knihovnou Simple. Knihovna Simple na základě anotací vytvoří instance objektů `SecurityView` (z elementů `<securityView>`). V rámci vytváření instancí `SecurityView` knihovna vytvoří i vnořené objekty `ModelRefObj` z elementů `<refobj>`. Objekty `ModelRefObj` reprezentují reference na objekty datového modelu a potřebujeme je jako vstup do druhé fáze zpracování modelu.

Fragment 3.3: Příklad elementu reprezentujícího balíček modelu

```
<packages>
    <package>
        <name locale="en">GO Sales (analysis)</name>
        <description locale="en">Package based on dimensional view.</description>
        <lastChanged>2012-05-25T13:26:40</lastChanged>
        <lastChangedBy>Anonymous</lastChangedBy>
        <screenTip locale="en"/>
        <lastPublished>2011-12-01T11:46:52</lastPublished>
        <lastPublishedCMPath>/content</lastPublishedCMPath>
        <definition>
            <viewref>[].[securityViews].[GO Sales (analysis)]</viewref>
        </definition>
```

```
        ...
    </package>
```

Fragment 3.4: Příklad elementu specifikujícího obsah balíčku modelu

```
<securityViews>
    <securityView>
        <name>GO Sales (analysis)</name>
        <definition>
            <set includeRule="include">
                <refobj>[go_sales]</refobj>
                <refobj>[Filters and calculations].[Returns - unsatisfactory or defective]</refobj>
                <refobj>[Filters and calculations].[Returns - damaged or wrong shipment]</refobj>
                <refobj>[Sales (analysis)].[Margin]</refobj>
                <refobj>[Filters and calculations].[Order value]</refobj>
                ...
```

Na konci první fáze zpracování modelu máme objekty `ModelStructure`, `PackageStructure` a `ModelRefObj`, které reprezentující částečně zpracovaný datový model. Ten jsme získali zpracováním dokumentu s XML specifikací datového modelu.

Ve druhé fázi získáme a zpracujeme informaci o lineage objektů modelu. Zpracováním lineage získáme úplné informace o objektech modelu, ke kterým zatím známe pouze jejich identifikátory (vyjádřené objekty `ModelRefObj`).

3.7 Získání a zpracování lineage objektů modelu

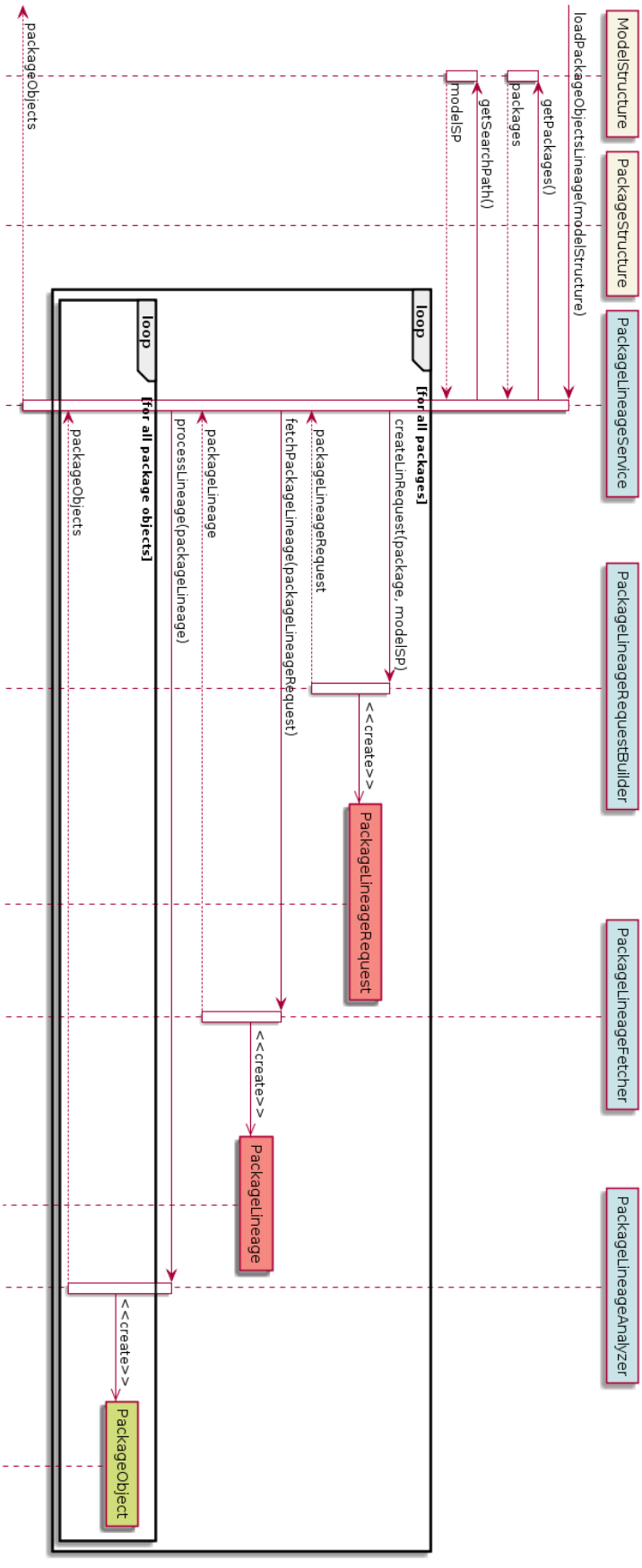
Z pohledu API IBM Cognos získání lineage objektů modelu spočívá v zaslání požadavku na lineage do Metadata Service. Požadavek na lineage je XML dokument, který je potřeba manuálně zkonstruovat. Metadata Service následně odpoví XML dokumentem se specifikací lineage.

Za získání a zpracování lineage objektů modelu je zodpovědná služba `PackageLineageService`. Diagram 3.7 ilustruje její fungování. Na jejím vstupu je objekt `ModelStructure`, který je zpracován po jednotlivých balíčcích modelu. Zpracování balíčku je rozděleno mezi tři služby konektoru.

`PackageLineageRequestBuilder` vygeneruje požadavek na lineage pro Cognos API. Požadavek je reprezentován objektem `PackageLineageRequest`. Tento objekt pouze zapouzdřuje XML dokument s požadavkem.

Služba `PackageLineageFetcher` pošle požadavek na lineage službě *Metadata Service* platformy Cognos. Odpověď s popisem lineage objektů je z Cognos API vrácena ve formě XML dokumentu. `PackageLineageFetcher` tuto odpověď zapouzdří do objektu `PackageLineage`.

Služba `PackageLineageAnalyzer` přijímá na vstupu lineage ve formě objektu `PackageLineage` a po zpracování lineage vrací kolekci objektů `PackageObject`, které definují objekty datového modelu (viz 3.5).



Obrázek 3.7: Získání a zpracování lineage objektů modelu

3.7.1 XML požadavek na lineage objektů modelu

Úkolem služby `PackageLineageRequestBuilder` je vytvořit požadavek na lineage objektů modelu. Požadavek je XML dokument, specifikovaný v rámci API Metadata Service. Ukázka 3.5 představuje požadavek na lineage pro objekt `[Sales (query)].[Products]` z balíčku `GO Sales (query)`.

Požadavek se skládá z elementu `<connection>` a jednoho či více elementů `<objectQuery>`. Element `<connection>` specifikuje v parametru `searchPath` cestu k objektu modelu pomocí search path. Cestu k modelu lze získat z objektu API `Report`. Elementy `<objectQuery>` specifikují dotazy na jednotlivé objekty modelu. Parametr `id` element `<objectQuery>` určuje pořadové číslo dotazu za účelem spárování s odpovědí na lineage dotaz. Element `<param>` specifikuje dotaz na objekt, v atributu `name` se určuje způsob specifikace dotazu. V případě dotazu na objekty modelu specifikujeme dotaz pomocí *objectId* objektu modelu. Tomu odpovídá hodnota `modelItemRef` atributu `name` elementu `<param>`.

Fragment 3.5: Příklad XML požadavku na lineage jednoho objektu modelu

```
<lineageRequest version='0.1'>
    <connection>
        <param name='searchPath'>/content/folder[@name='Packages']
            /package[@name='GO Sales (query)']/model</param>
    </connection>
    <objectQuery id='1'>
        <param name='modelItemRef'>[Sales (query)].[Products]</param>
    </objectQuery>
</lineageRequest>
```

V diagramu 3.6 vidíme, že `PackageLineageRequestBuilder` vytváří požadavek na základě search path modelu (získané z objektu `ModelStructure`) a struktury balíčku modelu (objekt `PackageStructure`. `PackageStructure` obsahuje reference na objekty modelu (objekty `ModelRefObj`). Tyto referencí představují hodnoty identikátorů (*objectId*) objektů modelu. Identifikátory objektů modelu `PackageLineageRequestBuilder` potřebuje pro vybudování jednotlivých elementů `<objectQuery>` pro XML dokument dotazu na lineage. Hodnotu search path použijeme pro vytvoření elementu `<connection>`.

3.7.2 Zaslání požadavku na lineage

Rozhraní `PackageLineageFetcher` má na základě požadavku na lineage získat odpověď s lineage. V rámci této úlohy se komunikuje se službou *Metadata Service* Cognos API, což odkrývá jedno z nejslabších míst Cognos API. [2] uvádí:

> „When you call a method on a service, the response header for the service may include important information that is required for subsequent requests. Some method calls require information from

> a previous call to the service, such as tracking information that indicates which server handled the previous request. For example, a `wait()` request must be routed to the same server that handled the primary request.
>
> When writing a Java application, when a method requires the header to include information from the return header of a previous method call, you must include code to copy the response header into the bibus » biBusHeader object. Before each method call on a service, you should update the header based on the response header value from the previous call. After retrieving the response header, clear the headers before setting the bibus » biBusHeader to the value of the response header.
>
> **Note:** The .NET toolkit manages the service headers automatically for C# applications.“

Z tohoto plyne, že Cognos API pro platformu JVM obsahuje oproti API pro platformu .NET poměrně zásadní nedostatek. Vyžaduje, aby uživatel API manuálně udržoval konverzaci se službami kopírováním hlaviček z odpovědí služeb do požadavků na služby. Při manipulaci s hlavičkami se odhaluje interní implementace služeb Cognos, realizovaná *SOAP* zprávami. To se týká jak práce s asynchronními metodami služeb (viz 1.4.4), tak volání několik různých služeb. Přihlášení do platformy se provádí proti službě Content Manager Service, ale pro následné volání Metadata Service je nutné provést manuální udržování konverzace.

V rámci práce vznikl kód pro volání Metadata Service, který se nepodařilo odladit ke spolehlivosti. Proto byl tento kód nahrazen vzorovým kódem IBM – třídou `RunLineageSpecification`, dostupnou z [6]. Ten zajišťuje manuální udržování konverzace a umožňuje pokládat dotazy na lineage. V rámci času vymezeného na implementaci konektoru se nestihlo kód nahradit vlastní implementací. Finální implementace služby `PackageLineageFetcher` je tedy realizována voláním převzaté třídy `RunLineageSpecification`.

3.7.3 XML lineage objektů modelu (odpověď na lineage požadavek)

Fragment 3.6: Příklad XML lineage jednoho objektu modelu

```
<lineageResponse>
    <queryResult>
        <objectRef>[Sales (query)].[Products]</objectRef>
        <objectQueryRef>1</objectQueryRef>
    </queryResult>
    <object id="/content/folder[@name='Packages']/package[@name='GO Sales (query)']">
       <name>GO Sales (query)</name>
        <type>package</type>
```

```
        ...
    </object>
    <object id="[Sales (query)]">
        <name>Sales (query)</name>
        <type>namespace</type>
        <property name="objectId" displayName="ID">[Sales (query)]</property>
        <property name="objectName" displayName="Name">Sales (query)</property>
        <property name="objectType" displayName="Type">Namespace</property>
        <parentRef>/content/folder[@name='Packages']/package[@name='GO Sales (query)']</parentRef>
        <childRef>[Sales (query)].[Products]</childRef>
    </object>
    <object id="[Sales (query)].[Products]">
        <name>Products</name>
        <type>querySubject</type>
        ...
    </object>
</lienageResponse>
```

Fragment 3.6 ukazuje XML dokument lineage, který vznikl jako odpověď na lineage požadavek na objekt modelu `[Sales (query)].[Products]` (pro odpovídající XML dokument požadavku viz sekce 3.7.1). Lineage dokument se skládá z elementů `<queryResult>` a `<object>`.

Každý element `<queryResult>` odpodvídá jednomu elementu `<objectQuery>` z XML dokumentu pro požadavek na lineage. `<queryResult>` obsahuje element `<objectQueryRef>` s pořadovým číslo objektu `<objectQuery>` z lineage požadavku. `<queryResult>` dále obsahuje element `<objectRef>`, který opakuje referenci na objekt modelu, který byl také součástí příslušné `<objectQuery>` z lineage požadavku.

Element `<object>` vyjadřuje specifikaci jednoho objektu datového modelu. Objekt je popsán elementy `<name>`, `<type>` a elementem `<property>`. Element `<name>` definuje název objektu, element `<type>` definuje typ objektu a element `<property>` reprezentuje „netypovanou“ vlastnost objektu (jednotlivé vlastnosti se liší hodnotou atributu `name` tohoto elementu).

Element `<object>` dále obsahuje elementy pro popis vztahů s jinými objekty. Elementy `parentRef` a `childRef` definují referenci na rodičovský objekt a reference na objekty – děti. Element `<transformation>` definuje transformační zdroje objektu. Transformací se rozumí předpis, na jehož základě vznikají data tohoto objektu z dat jiných objektů modelu.

Fragment 3.7: Příklad elementu <transformation> lineage XML

```
<object id="[Sales (query)].[Products].[Product line code]">
        <name>Product line code</name>
        <type>queryItem</type>
        ...
        <transformation>
            <type>objectReference</type>
            <property name="expression" displayName="Expression">
                [gosales].[PRODUCT_LINE].[PRODUCT_LINE_CODE]</property>
            <objectRef>[gosales].[PRODUCT_LINE].[PRODUCT_LINE_CODE]</objectRef>
```

```
    </transformation>
    <parentRef>[Sales (query)].[Products]</parentRef>
</object>
```

Fragment 3.7 zobrazuje objekt lineage, obsahující element `<transformation>` (pro stručnost jsou vynechány elementy `<property>`). Vidíme, že element `<transformation>` obsahuje element `<objectRef>`, který obsahuje referenci na jiný objekt modelu a představuje transformační zdroj. Elementů `<objectRef>` může být v rámci definice jedné transformace několik.

[2] uvádí, že definice transformace může obsahovat mimo referencí objektů (element `<objectRef>`) i definici požadavku na lineage. Zpracováním tohoto požadavku bychom získali definici objektu. Element `<transformation>` může tedy obsahovat i element `<lineageRequest>`. V rámci práce tato varianta transformace není podporována a v případě, že se vyskytne, bude ignorována.

Objekty, které jiný objekt referencuje jako svého rodiče (`<parentRef>`), dítě (`<childRef>`) nebo zdroj transformace (`<objectRef>` v `<transformation>`) jsou automaticky obsaženy v lineage dokumentu.

3.7.4 Zpracování XML lineage objektů modelu

V diagramu 1.11 analytického modelu jsme definovali, že z pohledu katalogu reportů nás zajímají závislosti objektů modelu (*Package Object*) mezi sebou a závislost objektů modelu na sloupci databáze (*Column*).

Diagram návrhu objektů loaderu 3.5 závislost mezi objektem datového modelu (třída `PackageObject`) a sloupci databáze (třída `TableColumn`) vyjadřuje pomocí vazby `usesDataFrom`. Závislost objektů datového modelu mezi sebou zde není zachycena, protože tuto vazbu zrealizujeme až na úrovni nahrávací úlohy platformy XF3 (podrobněji viz kapitola 4).

Postup zpracování XML dokumentu lineage je následující:

1. Z elementů `<queryResult>` získáme z podelementů `<objectRef>` identifikátory objektů modelu, které specifikoval lineage dotaz (nazýváme je jako primární objekty).
2. K seznamu identifikátorů primárních objektů přidáme identifikátory objektů, které jsou potomky primárních objektů. Vzniklý seznam označujeme jako seznam sekundárních objektů.
3. Pro každý sekundární objekt:
   a) Získáme informace o objektu, dostupné přímo z elementu, který jej popisuje.
   b) Položíme rekurzivní dotaz na datový zdroj objektu.
   c) Transformujeme získané informace o objektu do vhodné XML struktury.
4. Zpracujeme XML výsledek na straně loaderu v Javě (vytvoříme instance tříd `PackageObject`).

Vidíme, že pro zpracování lineage XML dokumentu nám nestačí specifikovat jednotlivé uzly dokumentu pomocí jazyka XPath. Bude potřebovat dotazovací a transformační schopnosti, které nabízí jazyk XQuery. [7] definuje XQuery jako dotazovací jazyk, umožňující výběr XML elementů z dokumentu, jejich reorganizaci a transformaci do zvolené XML struktury.

3.7.4.1 Dotaz na datový zdroj objektu

Jako ukázku XQuery použijeme funkci 3.8 pro získání datového zdroje objektu modelu. Funkce se nazývá `extract-sources` a náleží do jmenného prostoru `mlineage`. Přijímá parametr `$obj` typu uzel `<object>`, který odpovídá jednomu objektu modelu z lineage dokumentu. V parametru `$lineage` je předán uzel s celým dokumentem lineage, jehož kořenový element je `<lineageResponse>`. Funkce vrací `0-n` uzlů `<package-object>`.

Fragment 3.8: Rekurzivní funkce pro získání datových zdrojů objektu

```
declare function mlineage:extract-sources($obj as element(object),
                                          $lineage as element(lineageResponse))
as element(package-object)*
{
    if ($obj/type/text() = 'queryItem' and $obj/property[@name = 'externalName']) then
        mlineage:extract-query-item-specs($obj, $lineage)
    else
        for $transSourceId in $obj/transformation/objectRef/text()
        let $transSource := $lineage//object[@id = $transSourceId]
        return mlineage:extract-sources($transSource, $lineage)
};
```

Pokud je objekt modelu typu `queryItem` a obsahuje vlastnost s názvem `externalName`, která indikuje, že objekt má přímou transformaci vedoucí na externí datový zdroj, zpracujeme jej pomocí funkce `extract-query-item-specs`. V opačném případě zavoláme funkci rekurzivně na transformačních zdrojích objektu.

Fragment 3.9: Funkce pro získání vlastností objektu typu `queryItem`

```
declare function mlineage:extract-query-item-specs($qi as element(object),
                                                   $lineage as element(lineageResponse))
as element(data-source)
{
    <data-source>
        <column-name>{$qi/property[@name = 'externalName']/text()}</column-name>
        <database>{
                let $dsId := $qi/transformation/objectRef/text()
                let $dsNode := $lineage//object[@id = $dsId]
                let $schema := $dsNode/property[@name = 'schema']/text()
                return $schema
        }</database>
        <table-sql>{
                let $parentId := $qi/parentRef/text()
                let $parentNode := $lineage//object[@id = $parentId]
```

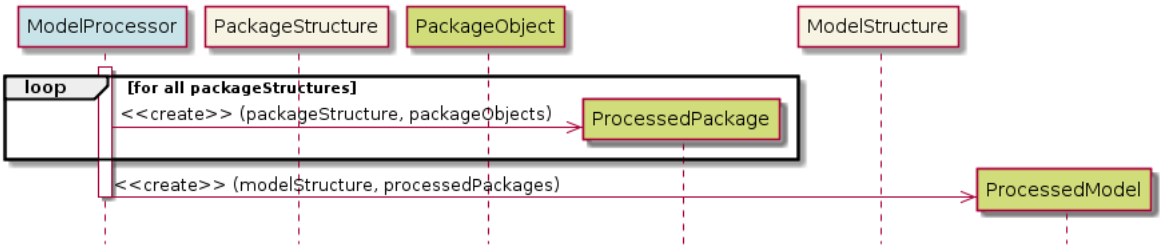
```
                let $sql := $parentNode/property[@name = 'sql']/text()
                return $sql
        }</table-sql>
        { mlineage:extract-aggregation($qi) }
        { mlineage:extract-data-type($qi) }
    </data-source>
};
```

Funkce `extract-query-item-specs` (3.9) získává informaci o relačním datovém zdroji objektu *queryItem*. Informaci o názvu sloupce (`<column-name`) funkce získá z hodnoty vlastnosti `externalName`. Název schématu (`<database>`) získá z objektu typu `dataSource`, na který vede reference transformace zpracovávaného objektu *queryItem*. SQL dotaz (`<table-sql>`), kterým se získají data pro tento objekt *queryItem*, získáme z rodičovského objektu, který je typu *querySubject*. Ten obsahuje element `<property>`, který obsahuje definici SQL dotazu. Následuje zpracování dalších dvou elementů `<property>`.

Fragment 3.10 zobrazuje XML strukturu pro vrácení výsledku z XQuery modulu `mlineage` pro zpracování lineage modelu. Vidíme dva zpracované objekty – objekt s názvem `Product details` typu úroveň hierarchie (*level*) a objekt s názvem `Product number` typu *queryItem*. Objekt úrovně hierarchie nemá žádné datové zdroje, objekt *queryItem* má jeden. Při srovnání identifikátorů obou objektů vidíme, že objekt `Product details` je rodičem objektu `Product number` (jeden identifikátor je podřetězcem druhého). Tento vztah není ve výsledku explicitně zachycen, ale objekty jsou lexikograficky seřazeny podle svých identifikátorů, což nám umožní rekonstruovat vztahy rodič – dítě na úrovni nahrávací úlohy XF3.

Fragment 3.10: XML struktura pro výstup zpracování lineage

```
<visible-objects>
    <package-object>
        <id>[Inventory (analysis)].[Products].[Products].[Product details]</id>
        <name>Product details</name>
        <type>level</type>
    </package-object>
    <package-object>
        <id>[Inventory (analysis)].[Products].[Products].[Product details].[Product number]</id>
        <name>Product number</name>
        <type>queryItem</type>
        <data-type>int32</data-type>
        <data-source>
            <column-name>PRODUCT_NUMBER</column-name>
            <database>GOSALES</database>
            <table-sql>Select * from [gosales].PRODUCT</table-sql>
            <aggregation>count</aggregation>
            <data-type>int32</data-type>
        </data-source>
    </package-object>
    ...
</visible-objects>
```



Obrázek 3.8: Sloučení objektů výsledků podúloh modelu

Zpracováním XML s výsledkem na straně Javy získáme instance tříd `PackageObject` a `TableColumn`, které reprezentují kompletní informace o objektech datového modelu.

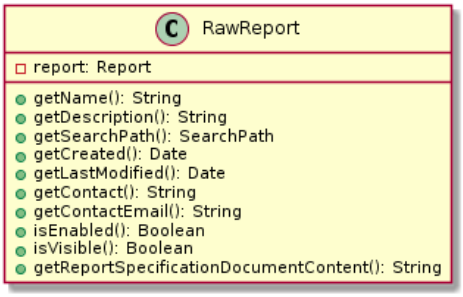
3.8 Sloučení výsledků podúloh zpracování modelu

Z první části zpracování modelu jsme získali strukturu modelu `ModelStructure`, která končí seznamem referencí na objekty modelu `ModelRefObj`. V druhé části jsme získali pro jednotlivé balíčky modelu seznamy objektů modelu (`PackageObject`) a jejich případné externí datové zdroje, reprezentované objektem `TableColumn`). Ten reprezentuje sloupec v tabulce databáze.

Pro schéma databáze, tabulku databáze a sloupec databáze nezavádíme samostatné objekty. Na objektu sloupce (`TableColumn`) držíme název schématu (atribut `database`) a i název tabulky (atribut `table`). Tím samozřejmě dochází k duplikaci, ke které bude docházet i na úrovni databáze. Objektu `TableColumn` odpovídá denormalizovaná tabulka `COC_TABLE_COLUMN`.

Tento návrh je motivován relativní složitostí získání informací o těchto objektech databázové vrstvy ve srovnání například se získáním informace o struktuře reportů. Objekty databázové vrstvy se v rámci lineage modelu, který je pro nás primárním zdrojem informací o objektech modelu, nevyskytují jako plnohodnotné entity. Získání informací o objektech databázové vrstvy je založeno na kombinování neúplných informací v rámci zpracování XML dokumentu lineage modelu. Zjednodušený objektový model založený na objektu `TableColumn` a příslušnou denormalizovanou tabulku jsme snadněji schopni naplnit získanými daty. Vytvoření samostatných entit typu schéma, tabulka a sloupec vyřešíme v rámci implementace odpovídajících aplikací na platformě XF3.

Diagram 3.8 zobrazuje proces sloučení objektů ze dvou fází zpracování. Při vytváření instance `ProcessedPackage` se kontroluje, zda seznam referencí na objekty modelu odpovídá seznamu dodaných objektů modelu.



Obrázek 3.9: Objekt RawReport, zapouzdřující objekt API

3.9 Přehled zpracování reportu

U reportu obdobně jako u modelu operujeme s několika typy abstrakcí. API platformy IBM Cognos poskytuje objekty `Report`. Ty v rámci loaderu zapouzdřujeme do objektu `RawReport`, který je vyobrazen na diagramu 3.9.

Stejně jako u modelu získáváme část informací o reportu z objektu API a část informací získáme zpracováním XML specifikace. Z objektu API (respektive ze zapouzdřujícího objektu `RawReport`) získáme metadata o reportu, jako je například datum poslední modifikace a některé základní údaje, jako je název reportu a jeho *search path*. Informace o struktuře reportu lze získát pouze zpracováním XML. XML specifikaci reportu poskytuje objekt `RawReport` pomocí metody `getReportSpecificationDocumentContent`.

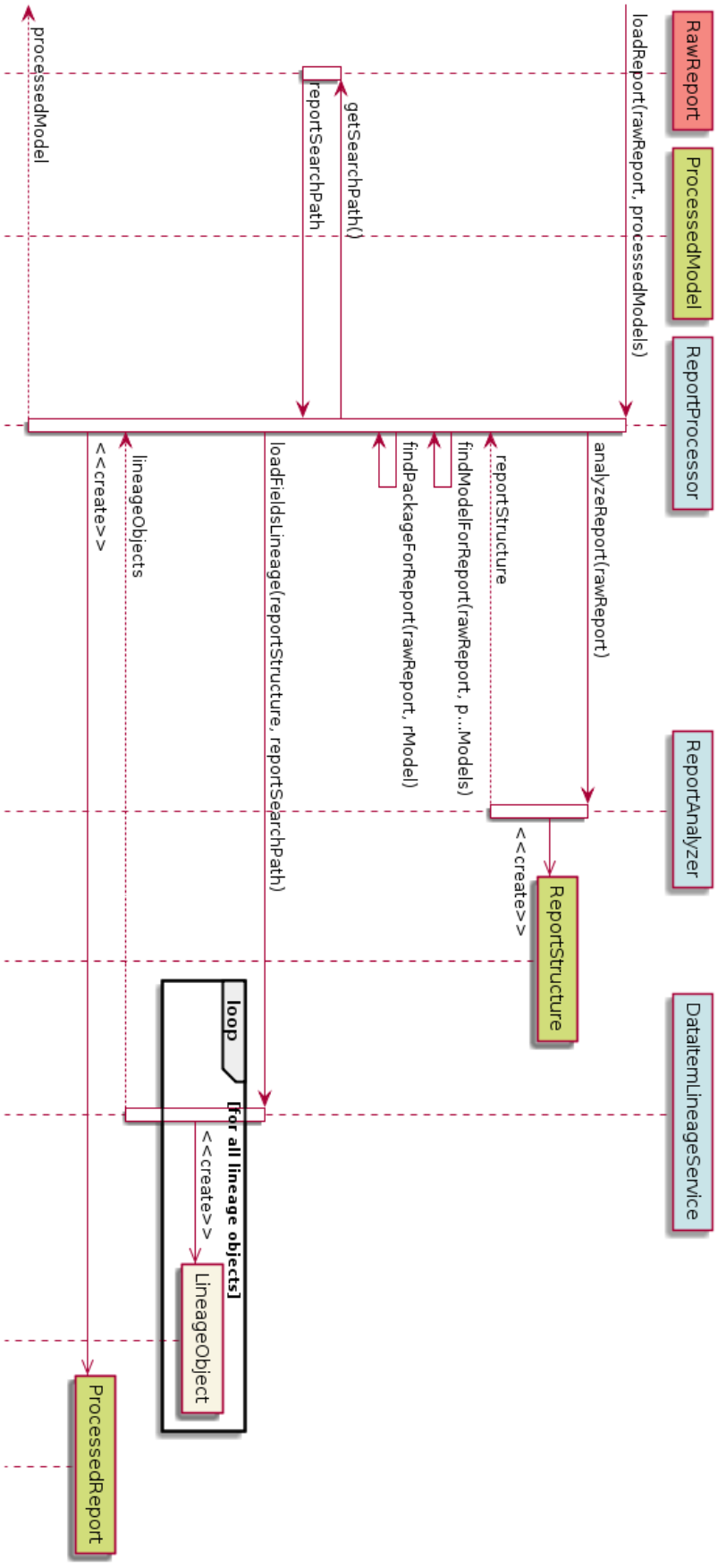
Sekvenční diagram 3.10 ilustruje, jak dochází k celkovému zpracování reportu. Na vstupu je objekt nezpracovaného reportu (objekt `RawReport`) a kolekce již zpracovaných modelů (objekty `ProcessedModel`). V první fázi vzniká objekt struktury reportu `ReportStructure` analýzou XML specifikace reportu, kterou provádí služba `ReportAnalyzer`. Objekt `ReportStructure` obsahuje mimo jiné informace o datových položkách reportu.

Na základě znalostech o datových položkách získáme pomocí služby `DataItemLineageService` informaci o data lineage – tom, na kterých objektech datového modelu závisí datové položky reportu. Tato informace je reprezentována objektem `LineageObject` a použijeme ji v poslední fázi zpracování k spárovaní datových položek (`DataItem` s objekty modelu (`PackageObject`).

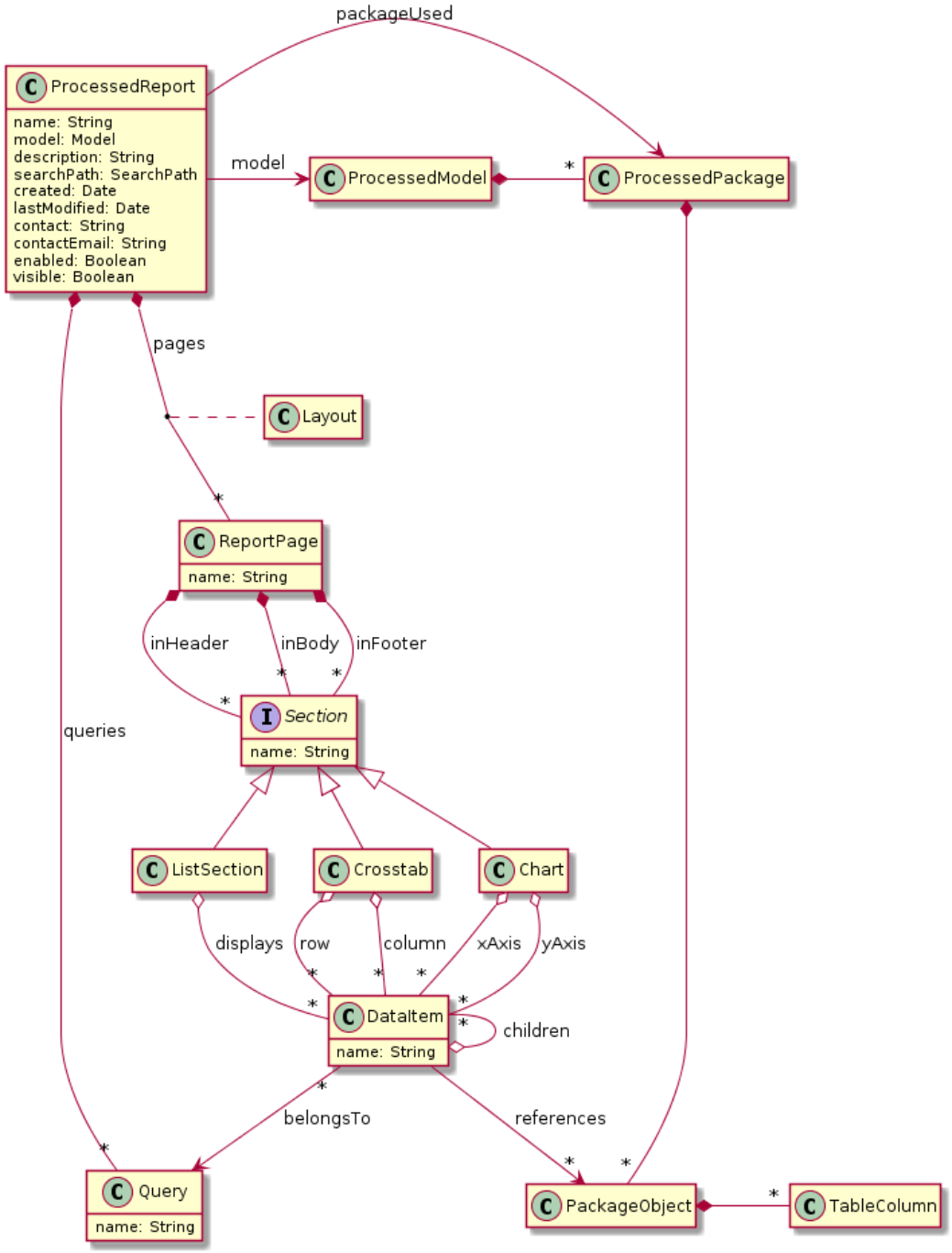
V konečné fázi dojde ke spárování reportu s modelem a příslušným balíčkem modelu. Dále dojde ke spárování datových položek reportu s objekty modelu. Kompletní zpracování reportu reprezentuje objekt `ProcessedReport`.

3.9.1 Struktura objektů pro zpracovaný report

Diagram 3.11 zachycuje strukturu objektů reprezentujících zpracovaný report. Samotný report představuje třída `ProcessedReport`. Report se odkazuje na objekt zpracovaného modelu `ProcessedModel`, což vyjadřuje vztah, že report



Obrázek 3.10: Sekvenční diagram zpracování objektu Report z API



Obrázek 3.11: Objekt zpracovaného reportu

používá data z modelu. Report se také odkazuje na konkrétní balíček modelu `ProcessedPackage`. To odpovídá tomu, že report používá z modelu data právě z jednoho balíčku. Objekt reportu dále obsahuje ty atributy, které jsme definovali na objektu nezpracovaného reportu `RawReport`.

Report může definovat několik vizuálních rozvržení (objekt `Layout`). Každý layout je složen ze stránek a neobsahuje žádnou dodatečnou informaci. Report obsahuje na logické úrovni report queries.

Stránka je vyjádřená objektem `ReportPage` a má název. Skládá se z hlavičky (*header*), těla (*body*) a patičky (*footer*).

V každé z těchto tří částí stránky může být blok (objekt `Block`), což je buď tabulka (`List`), kontingenční tabulka (`Crosstab`) nebo graf `Chart`. Blok má název a zobrazuje datové položky (objekt `DataItem`).

Datová položka má název a patří do report query (objekt `Query`), která datové položky logicky sdružuje a má interní název.

Kontingenční tabulka má řádky a sloupec. Jak řádek, tak sloupec zobrazují hierarchie datových položek (vztah `children` mezi datovými položkami).

Tabulka obsahuje jednu nebo více datových položek. V případě jedné datové položky je tabulka seznamem.

Graf obsahuje osy x a y, přičemž nemusí být přítomny obě osy. Na každé ose grafu může být jedna nebo více datových položek.

3.10 XML specifikace reportu

Objekt API `Report` poskytuje interní specifikaci reportu v podobě XML dokumentu. Fragment 3.11 ukazuje přehled podstatných částí XML specifikace reportu `2011 Sales Summary`. Z pohledu katalogu reportů je pro nás podstatný element `<layout>`, který definuje vizuální strukturu jednotlivých stránek reportu. Dále nás zajímá element `<queries>`, který obsahuje queries použité v reportu a informace o datových položkách, které tyto queries sdružují. Element `<XMLAttributes>` obsahuje informace technického charakteru, ty nechceme katalogizovat. Element `<reportVariables>` obsahuje proměnné reportu. Ty v katalogu reportů neplánujeme reprezentovat, jedná se o příliš detailní informaci.

Fragment 3.11: XML dokument specifikace reportu

```
<report xmlns="http://developer.cognos.com/schemas/report/9.0/"
        expressionLocale="en-us">
    <modelPath>/content/folder[@name='Packages']/...</modelPath>
    <layouts>
        <layout>
            <reportPages>
                <page name="Page1">
                    <pageBody>
                        <contents>
                            ...
```

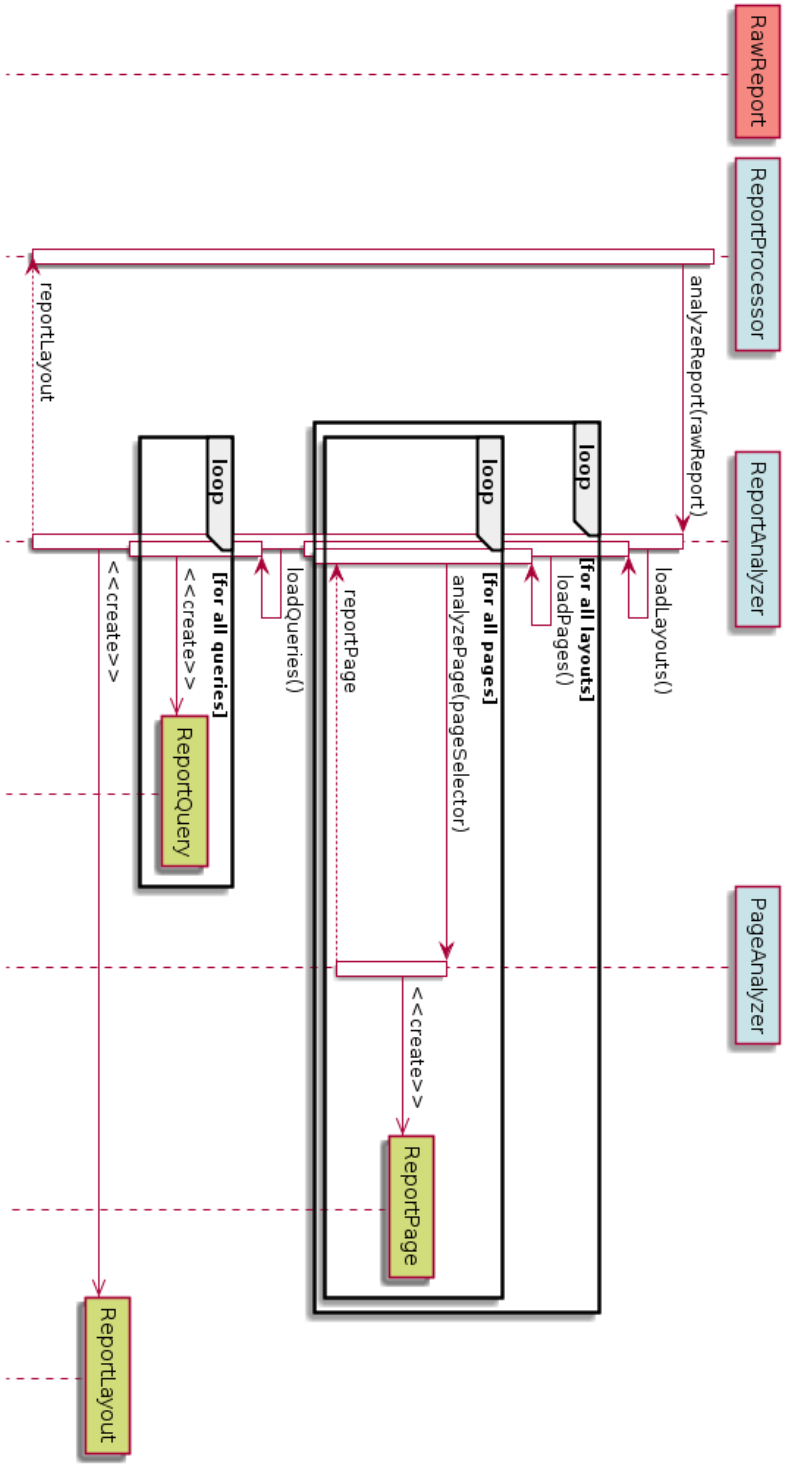
```
                    </contents>
                </pageBody>
                <pageHeader>
                    <contents>
                        ...
                    </contents>
                </pageHeader>
                <pageFooter>
                    <contents>
                        ...
                    </contents>
                </pageFooter>
                <style>...</style>
            </page>
            ...
        </reportPages>
    </layout>
</layouts>
<queries>
    <query name="Top 10 Sales Staff List">
        ...
    </query>
    ...
</queries>
<XMLAttributes>
    ...
</XMLAttributes>
<reportVariables>
    ...
</reportVariables>
<reportName>2011 Sales Summary</reportName>
</report>
```

3.11 Zpracování XML specifikace reportu

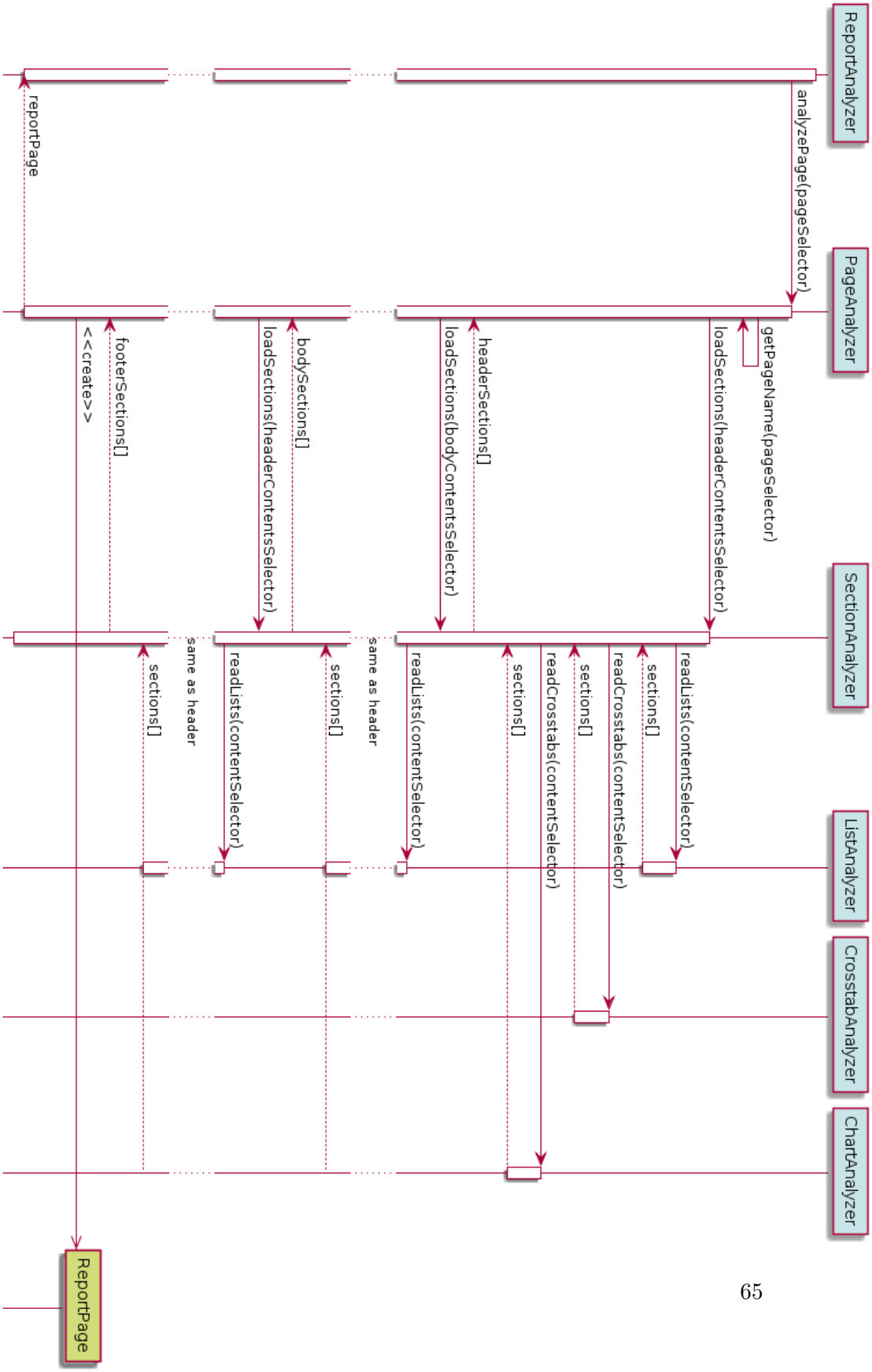
V rámci zpracování XML reportu používáme opět technologie XPath a XQuery s využitím stejných knihoven, jako při zpracování XML modelu (*Simple, XPath Selectors* a XML databáze *eXist*).

Za zpracování XML reportu odpovídá služba `ReportAnalyzer`. Ta v rámci metody `loadLayouts` zpracovává jednotlivé elementy `<layout>`, reprezentující vizuální rozložení. Element `<layout>` obsahuje pouze seznam stránek (elementy `<page>`). Zpracování stránek řeší služba `PageAnalyzer`. Ta dostane objekt, reprezentující XPath selektor ukazující na XML uzel konkrétní stránky a vrátí objekt `ReportPage`, představující zpracovanou stránku. Po zpracování všech stránek zpracujeme queries reportu. U query nás zajímá pouze její název, ten získáme pomocí XPath selektoru.

Fungování `PageAnalyzer` je vidět na sekvenčním diagramu 3.13. `PageAnalyzer` dostane na vstupu XPath selektor s uzlem stránky (element `<page>`) a pomocí dalšího XPath dotazu získá jméno stránky. Následně dojde ke zpracování uzlů



Obrázek 3.12: Sekvenční diagram zpracování XML specifikace reportu



Obrázek 3.13: Sekvenční diagram zpracování XML uzlů stránek reportu

jednotlivých sekcí reportu – hlavičky, těla a patičky. Všechny tři elementy odpovídající sekcím stránky obsahují podelement `<content>`, který obsahuje elementy jednotlivých bloků, kterými jsou `<crosstab>` pro kontingenční tabulku, `<list>` pro seznam a `<v2_combinationChart>` pro graf.

Zpracování elementu `<content>` je identické pro hlavičku, tělo i patičku reportu. To je naznačeno i v diagramu 3.13. V podstromech elementů `<content>` hledáme elementy odpovídající jednotlivým typům bloků (kontingenční tabulce, tabulce a seznamu). Za tímto účelem předkládáme uzel `<content>` službám `ListAnalyzer`, `CrosstabAnalyzer` a `ChartAnalyzer`.

Fragment 3.12: XML specifikace grafu na reportu

```
<v2_combinationChart maxHotspots="10000" name="Americas" refQuery="Sales Figures
        America Region Combination Chart">
    <v2_combinationTypeTooltips/>
    <v2_commonAxis>
        <v2_ordinalAxis>
            ...
        </v2_ordinalAxis>
        <chartNodes>
            <chartNode>
                <chartNodeMembers>
                    <chartNodeMember refDataItem="Quarter">
                        <chartContents>
                            <chartTextItem>
                                <dataSource>
                                    <memberCaption/>
                                </dataSource>
                            </chartTextItem>
                        </chartContents>
                    </chartNodeMember>
                </chartNodeMembers>
            </chartNode>
            ...
```

Fragment 3.12 zobrazuje část XML specifikace reportu, definující blok typu graf (element `<v2_combinationChart>`). Tento element obsahuje atribut `refQuery`, který obsahuje název query, kterou graf referencuje. Elementy `<list>` a `<crosstab>` také obsahují atribut `refQuery`. Ze všech typů bloků jsme tedy schopni získat název query, kterou blok používá.

Dále získáme zpracováním elementu `<v2_combinationChart>` názvy datových položek, které graf zobrazuje. Ty jsou hodnotami atributů `refDataItem`. Elementy `<list>` a `<crosstab>` obsahují atribut `refDataItem` na jiných elementech, ale vhodným XPath dotazem je jsme schopni získat ze všech typů bloků.

Informace o queries reportu je obsažena v samostatném elementu `<queries>`. Query reprezentuje dotaz do zdroje dat, typicky je zdrojem dat datový model. V rámci query jsou definovány jednotlivé datové položky. Definice datových položek obsahuje některé informace, například typ datové položky – zda je daná položka mírou, faktem atd. Problém se zpracováním těchto infor-

mací spočívá v tom, že pro datové položky neexistuje jednotná XML struktura. Setkáváme se s elementem `<dataItem>`, ale také `<dataItemLevelSet>` a `<dataItemMeasure>`, přičemž jednotlivé elementy mají zcela odlišnou strukturu.

Toto činí parsování uzlu specifikace reportu `<queries>` poměrně náročným. Proto jsme se rozhodli místo toho informace o datových položkách získávat pomocí výše zmíněného zpracování elementu `<content>` bloků reportu. Některé informace, jako například typ datové položky, následkem toho nebudeme mít k dispozici.

3.12 Zpracování lineage reportu

Diagram 3.11 struktury objektů zpracovaného reportu definuje vazbu `references` mezi datovou položkou reportu (`DataItem`) a objekty datového modelu (`PackageObject`). Ta říká, které objekty modelu datová položka využívá.

V sekci 1.3.6 jsme zavedli, že informaci odpovídající této vazbě nejsnáze získáme z lineage jednotlivých datových položek reportu. Další alternativou by bylo informaci získat z lineage pro celý report, ale to jsme pro vyšší náročnost zavrhnuli. V předchozí sekci jsme zavrhli možnost za účelem získaní této informace parsovat jednotlivé uzly `<query>` XML reportu.

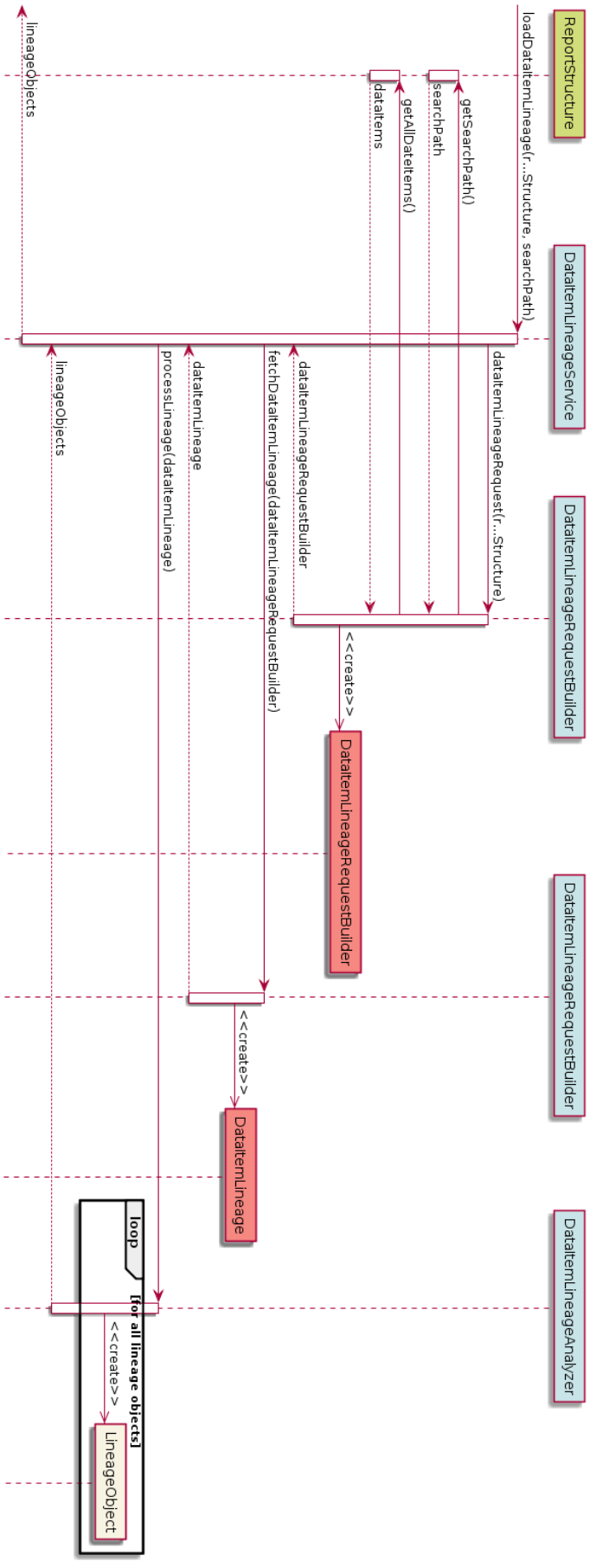
Diagram 3.14 ilustruje postup zpracování lineage datových položek reportu. Postup je velmi podobný se zpracováním lineage objektů modelu (viz diagram 3.14), proto se zaměříme se pouze na odlišnosti.

Služba `DataItemLineageRequestBuilder` využívá pro sestrojení požadavku na lineage objekt zpracovaného reportu `ProcessedReport`. Z toho objektu využije search path (která je vyžadována stejně jako v případě lineage modelu) a seznam datových položek reportu, z nichž jsou sestrojeny jednotlivé dotazy požadavku (elementy `<objectQuery>`).

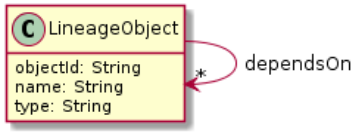
V sekci 1.3.6 bylo zmíněno, že využíváme služby *Content Manager Service* pro získání obou typů lineage. Proto není překvapující, že rozhraní `DataItemLineageFetcher` je implementováno stejnou třídou jako v případě rozhraní `PackageLineageFetcher`. Implementace předá službě Content Manager Service v obou případech XML stejného schématu, pouze s odlišnými hodnotami XML atributů.

Služba `DataItemLineageAnalyzer` zpracovává XML dokument, který má stejnou strukturu, jako v případě lineage objektů modelu (zpracovávaný službou `PackageLineageAnalyzer`). V případě lineage datových položek ale hierarchickou strukturu objektů lineage reprezentujeme rovnou v struktuře výsledku. To lze vidět na diagramu 3.15, který zobrazuje zpracovaný objekt lineage specifikace.

Posledním krokem zpracování lineage reportu je použití zpracovaných objektů lineage (`LineageObject`) k spárování datových položek reportu s objekty datového modelu.



Obrázek 3.14: Získání a zpracování lineage datových položek reportu



Obrázek 3.15: Zpracovaný objekt lineage datových položek

3.13 Celkový objektový model

Diagram 3.16 zobrazuje celkový objektový model výsledku zpracování reportů. Jde o sloučení diagramů 3.5 a 3.11. V následující kapitole se budeme zabývat uložením modelu do vhodného relačního schématu.

3.14 Databázové struktury pro uložení dat

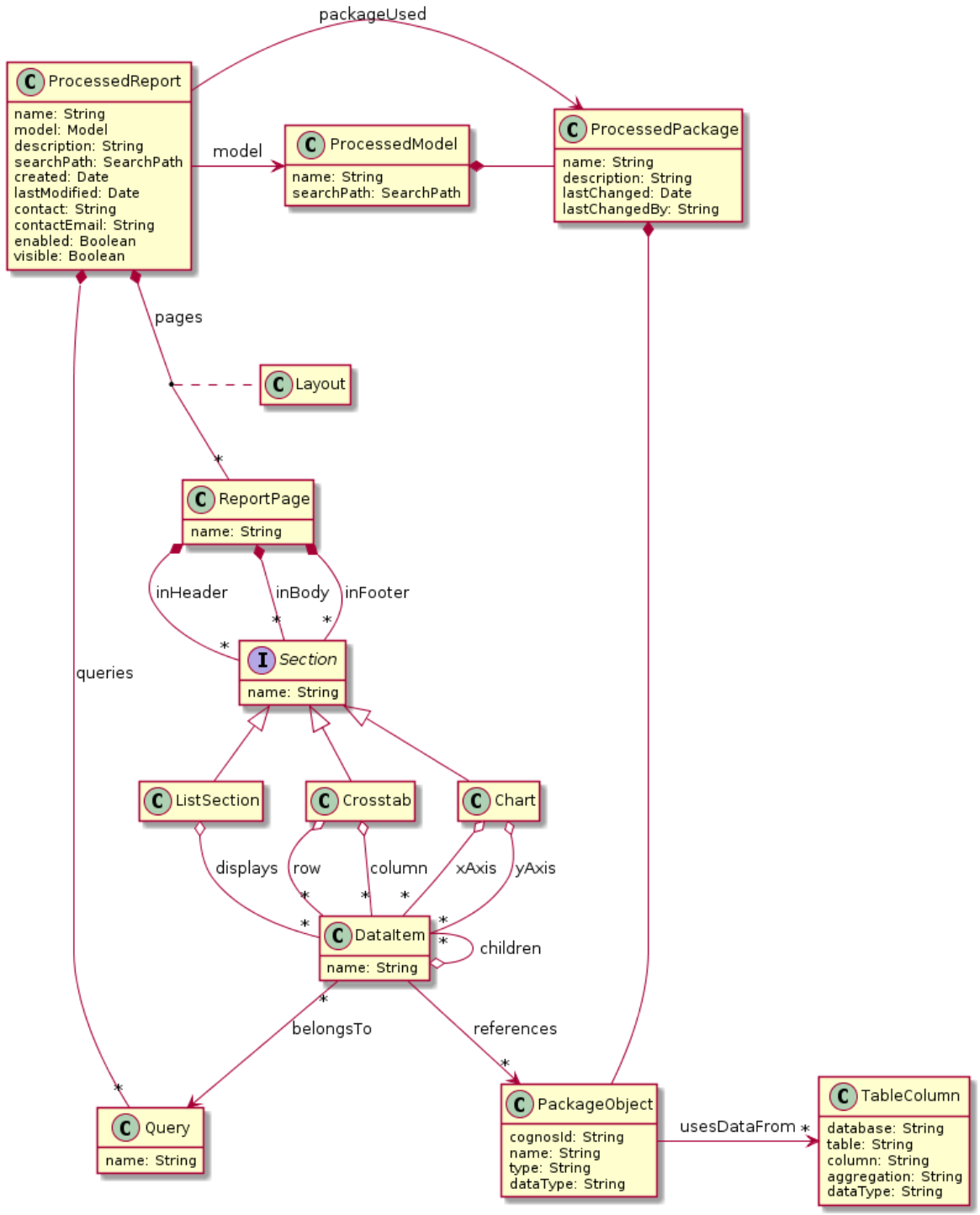
Diagram 3.17 zobrazuje relační schéma pro uložení získaných dat. Názvy u spojnic reprezentují cizí klíče. Z důvodu souladu s interním názvoslovím firmy Semanta mají databázové tabulky prefix `COC_` (zkratka z COgnos Connector).

Jako primární klíče tabulek používáme identifikátory generované na úrovni objektů v Javě. Pro toto definujeme rozhraní `HasId`. Typicky jsou jako identifikátory použity jména objektů, ale tuto závislost nechceme zavádět ve formě primárních klíčů nad sloupcem názvu objektu. Motivací pro toto je, aby se případná změna za jiný typ identifikátoru nedotkla databázového schématu. Databázové schéma používáme jako formu rozhraní mezi loaderem a aplikacemi na platformě XF3 a jeho změny vynucují příslušné změny na straně nahrávacích úloh XF3.

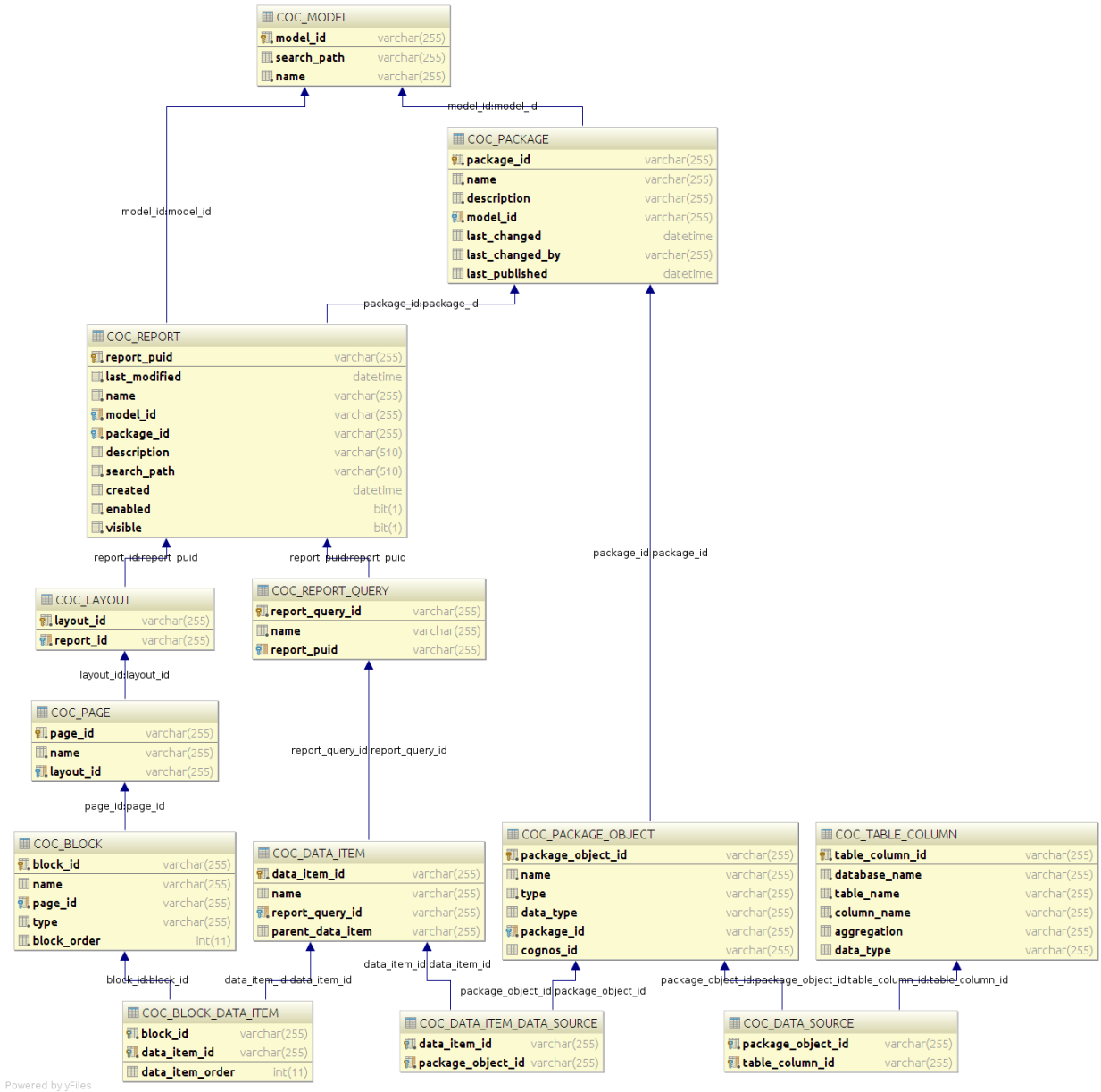
Objekt `TableColumn`, reprezentující entity z databázové vrstvy, mapujeme přímo na denormalizovanou tabulku (`COC_TABLE_COLUMN`). Objekty datového modelu `ProcessedModel`, balíčku modelu `ModelPackage` a objektu modelu `PackageObject` mapujeme přímo na tabulky `COC_MODEL`, `COC_PACKAGE` a `COC_PACKAGE_OBJECT`. Mezi objektem modelu a sloupcem databáze je vazba typu N–N, kterou na relační úrovni modelujeme vazební tabulkou `COC_DATA_SOURCE` mezi tabulkami `COC_TABLE_COLUMN` a `COC_PACKAGE_OBJECT`.

Objekt `ProcessedReport` mapujeme na tabulku `COC_REPORT`. Objekt query reportu `Query` mapujeme na tabulku `COC_REPORT_QUERY`. Objekt, reprezentující datovou položku reportu (`DataItem`) mapujeme na tabulku `COC_DATA_ITEM`, která má cizí klíč do tabulky pro vyjádření příslušnosti s tabulkou `COC_REPORT_QUERY`. Datovou závislost datové položky na balíčcích objektu, což je vztah typu N–N, reprezentujeme vazební tabulkou `COC_DATA_ITEM_DATA_SOURCE`.

Rozložení stránky (objekt `Layout`) mapujeme na tabulku `COC_LAYOUT`, objekt stránky reportu (`ReportPage`) na tabulku `COC_PAGE`. Identifikátor layoutu (`layout_id`), který je primárním klíčem tabulky, vzniká na úrovni loaderu



Obrázek 3.16: Třídní diagram objektového modelu zpracovaného reportu



Obrázek 3.17: Databázové schéma pro uložení dat

z atributů stránek a nepřináší žádnou novou informaci. Proto by bylo možné alternativně mapovat objekt `Layout` na vazební tabulku, která by měla složený primární klíč.

3.14.1 Redukce ukládaných informací

Situace je složitější u objektu bloku reportu `Block`. Historicky se měl databázový výstup této práce snažit sjednotit s podobným *konektorem* pro platformu *Business Objects*. Protože konektor pro Business Objects ukládá informace o bloku reportu jednotně bez ohledu na type bloku, byly detailní informace o jednotlivých typech bloků v Cognos konektoru při zápisu do databáze redukovány.

Jednotlivé typy bloků – graf (`Chart`), tabulka (`ListSection`) a kontingenční tabulka (`Crosstab`) – jsou tedy mapovány na tabulku `COC_BLOCK`, která reprezentuje obecný blok reportu. Typ bloku je vyjádřen sloupcem `type`. Sloupec `block_order` vyjadřuje pořadí bloku na stránce. Pořadí získáme z indexu objektu reprezentujícího blok v příslušné kolekci na objektu `ProcessedReport`.

3.15 Shrnutí

V rámci této kapitoly jsme popsali implementaci loaderu, který na vstupu získá cestu k reportům a na výstupu zapíše data o reportech a přidružených objektech do relační databáze. V následující kapitole si popíšeme implementaci aplikací na platformě XF3, které slouží k prezentaci dat. Součástí je i popis nahrávací úlohy, která používá data v databázi jako zdroj pro vytvoření obsahu jednotlivých aplikací.

Kapitola 4

Návrh prezentace metadat

Na konci kapitoly 3 jsme popsali databázové schéma pro uložení dat, která získáváme loaderem z platformy IBM Cognos. V rámci této kapitoly popíšeme finální část práce, která řeší implementaci katalogu reportů pro zobrazení získaných dat. Katalog je implementován na platformě XF3. Budeme se opírat o vlastnosti platformy, které jsme detailně popsali v kapitole 2.

Implementace katalogu reportů se skládá z XF3 aplikací pro jednotlivé typy objektů platformy Cognos a dále z definice nahrávací úlohy. Ta předepisuje, jak z dat uložených v relační databázi mají být vytvořeny objekty (*entry*) jednotlivých aplikací.

Databázové schéma, nadefinované v sekci 3.14, slouží jako rozhraní mezi loaderem a implementací katalogu v XF3. Proto budeme při návrhu aplikací a definici nahrávací úlohy vycházet z diagramu 3.17.

4.1 Redukce prezentovaných informací

Při definici aplikací katalogu reportů jsme se rozhodli nezavádět aplikaci pro všechny typy objektů, které ukládáme do tabulek v databázi. Velké množství typů aplikací snižuje přehlednost výsledného řešení.

V minulé kapitole jsme zmínili, že rozložení reportu (tabulka `COC_LAYOUT`) o sobě neposkytuje žádné dodatečné informace mimo seznamu stránek, které do něj patří. Všechny demonstrační reporty dodávané s platformou IBM Cognos navíc obsahují pouze jedno rozložení. V rámci přizpůsobení prezentace dostupným datům budeme rozložení reportu zanedbávat a budeme předpokládat v každém reportu právě jedno rozložení. Nebudeme tedy pro rozložení zavádět samostatnou aplikaci. Dodatečné přidání podpory pro více rozložení je triviální. Jednalo by se o vytvoření jedné definice aplikace a malou úpravu dvou existujících definic a malou změnu nahrávací úlohy.

Tabulka `COC_PAGE` reprezentuje informaci, na které stránce reportu se nachází který blok reportu. Jediná informace o stránce kromě seznamu bloků na ní je název stránky. Pokud bude mít konkrétní zpracovaný report malý počet stran (1–2), pak neponese entry stránky v prezentaci téměř žádnou přidanou hodnotu, naopak bude komplikovat navigaci entry reportu. Stránku (resp. její název) proto budeme reprezentovat jako další field entry *Block*.

Tabulka `COC_REPORT_QUERY` definuje query – technickou informaci o logickém seskupení datových položek reportu. Dále evidujeme vizuální seskupení datových položek do bloků reportu (vazební tabulka `COC_BLOCK_DATA_ITEM`). Pokud bychom v prezentaci reportu zobrazovali 2 typy seskupení datových položek (tedy kromě seznamu bloků reportu i seznam jeho queries), dostaneme dvě možné cesty, kudy se uživatel může donavigovat na entry datové položky z entry reportu. Ukázalo se, že tato varianta snižuje přehlednost výsledku. Proto jsme se rozhodli reprezentovat query skrze její název jako field entry *Data Item*.

4.2 Aplikace metadata katalogu

Diagram 4.1 zobrazuje jednotlivé aplikace, které vznikly na základě databázového schématu s použitím výše uvedených zjednodušení.

V analytickém modelu v kapitole 1 rozlišujeme mezi objekty vazbu typu kompozice a vazbu typu reference. U reference dále rozlišujeme, zda se jedná o jednosměrnou či obousměrnou vazbu.

Pro kompozici existuje na úrovni aplikací XF3 odpovídající konstrukt, kterým je field typu `<sub-application>`. Ten zakládá mezi dvěma aplikacemi hierarchickou vazbu rodič – dítě.

Vazbu typu reference zrealizujeme použitím fieldu typu `<relations>` na aplikaci, ze kterého vazba vychází. Obousměrnou vazbu zrealizujeme přidáním opačného směru vazby použitím fieldu typu `<lucene-query>` na cílové aplikaci. Oba typy fieldů jsme blíže popsali v kapitole 2.

4.2.1 Aplikace Report

Obrázek 4.2 zobrazuje entry aplikace Report.

V levé horní skupině vidíme název reportu, jeho search path a interní popis reportu, jak se zobrazuje v IBM Cognos.

V levé dolní skupině je seznam bloků reportu. V pravé horní skupině je skupina, kterou vyplňuje uživatel XF3. Je zde uživatelský popis reportu a uživatelem dodaný screenshot reportu.

Vpravo uprostřed jsou metadata o reportu: datum vytvoření, datum poslední modifikace, viditelnost reportu a příznak, zda je report aktivní.

Skupina vpravo dole obsahuje informaci o balíčku datového modelu, který report používá jako zdroj dat. Vazba na entry balíčku je typu reference a je realizována fieldem typu `<relations>`, který jsme popsali v kapitole 2.

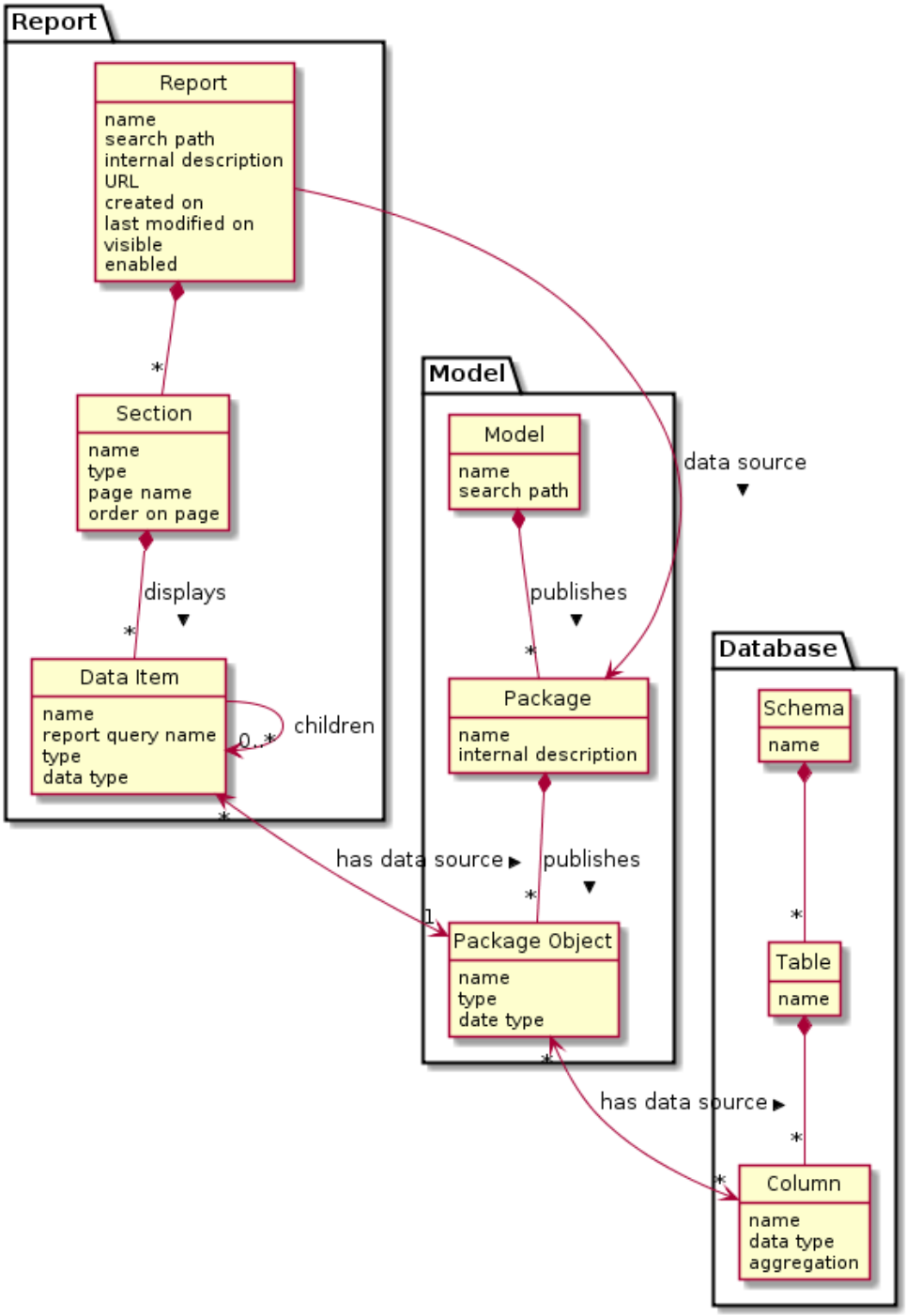
Aplikace Report implementuje roli `url-catalog`, což znamená, že entry typu Report jsou přístupná z komponenty Air. Všechny aspekty konfigurace komponenty Air vysvětlíme pohromadě v sekci 4.15.

4.2.2 Subaplikační field

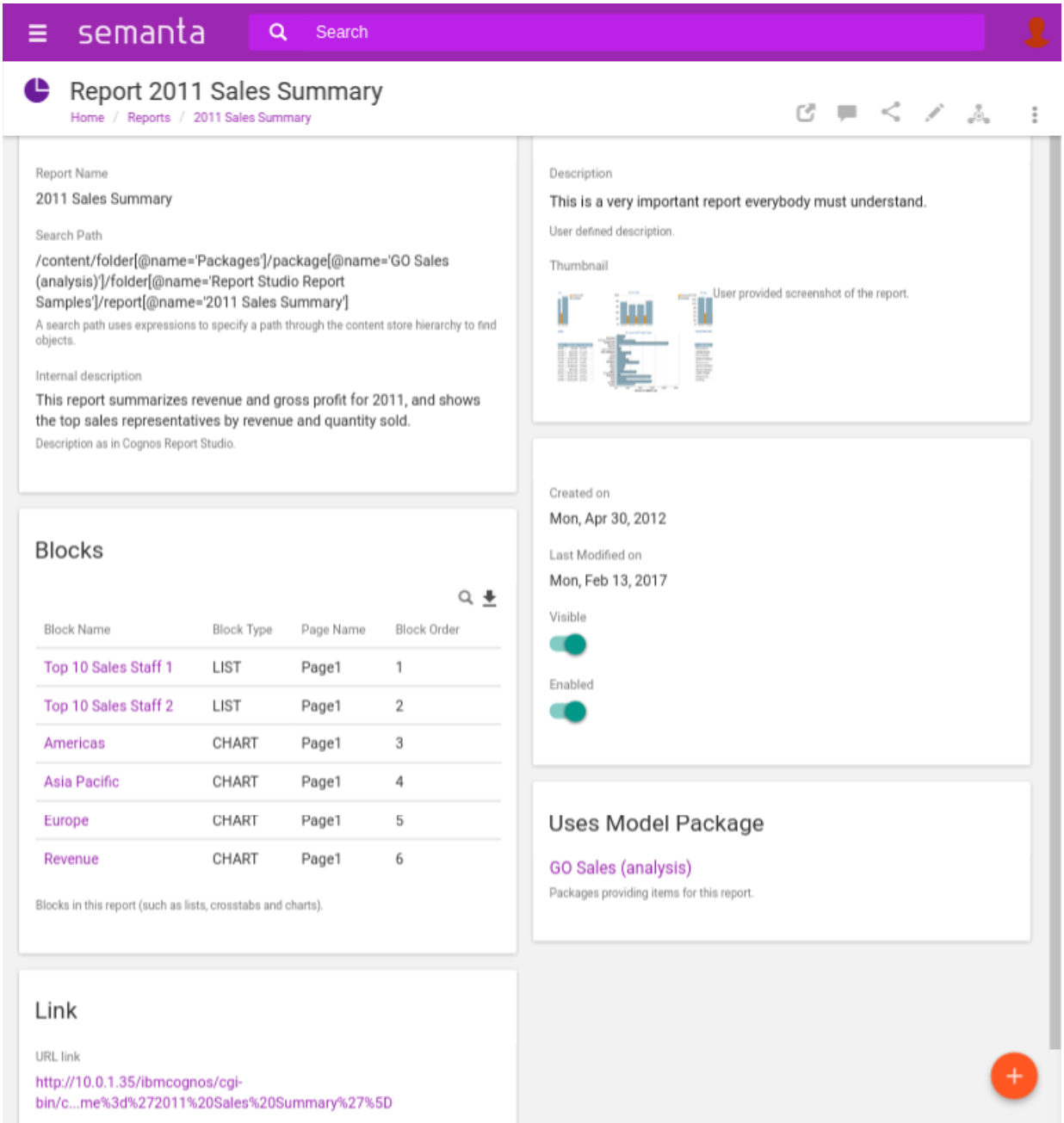
Field pro zobrazení bloků reportu je typu subaplikace, který jsme v kapitole 2 nedefinovali. Fragment 4.1 ukazuje definici tohoto fieldu v rámci aplikace report.

Fragment 4.1: Definice subaplikačního fieldu v aplikaci Report

```
<field>
    <code>sections</code>
    <name>Sections</name>
    <type>
        <sub-application>
            <app-ref-xid>coc-layout-page-section</app-ref-xid>
            <display-as>table</display-as>
```



Obrázek 4.1: Přehled aplikací metadat katalogu a vztahů mezi nimi



Obrázek 4.2: Entry aplikace Report

```
            <order-by>sectionOrder</order-by>
        </sub-application>
    </type>
    <hint-for-view>Sections in this report (such as lists, crosstabs and charts).</hint-for-view>
</field>
```

Field typu `<sub-application>` zavádí vztah rodič–dítě mezi dvěma aplikacemi. Aplikace, ve které je field definován, je rodičem. Druhá aplikace je referencována pomocí jejího id v elementu `<app-ref-xid>`. Element `<display-as>` nastavuje způsob, jakým budou entries – děti zobrazeny. Hodnota `table` volí jako typ zobrazení tabulku, která podporuje paginaci a řazení. Element `<order-by>` definuje, podle hodnoty kterého fieldu entries – dětí má v tabulce provedeno výchozí řazení. Seznam sloupců tabulky (fieldů entry – dítěte) je určen v rámci definice aplikace, která je dítětem. Způsob definice fieldů pro sloupce tabulky jsme popsali v sekci 2.2.2.1.

4.2.3 Rozšířený název, ikona

Na úrovni definice aplikace můžeme pomocí elementu `<extended-name>` nadefinovat rozšířený název entry. Jeho hodnota bude dynamicky vyhodnocena za použití Velocity s kontextem aktuálního entry. V případě aplikace Report definujeme jednoduché jméno entry jako `Report` a plný název jako `Report $entry.name`. Na předchozím obrázku 4.2 vidíme, že pokud je rozšířený název zadefinován, zobrazuje se v hlavičce entry místo povinného fieldu `entryName`.

Z obrázku 4.2 také vidíme, že entry reportu má v hlavičce entry vlastní ikonu. Tu definujeme pro celou aplikaci pomocí elementu `<icon>`, jehož obsahem je soubor s ikonou zakódovaný kódováním Base64 pro reprezentaci binárních dat v textové podobě.

Ukázka použití obou prvků je ve fragmentu 4.2.

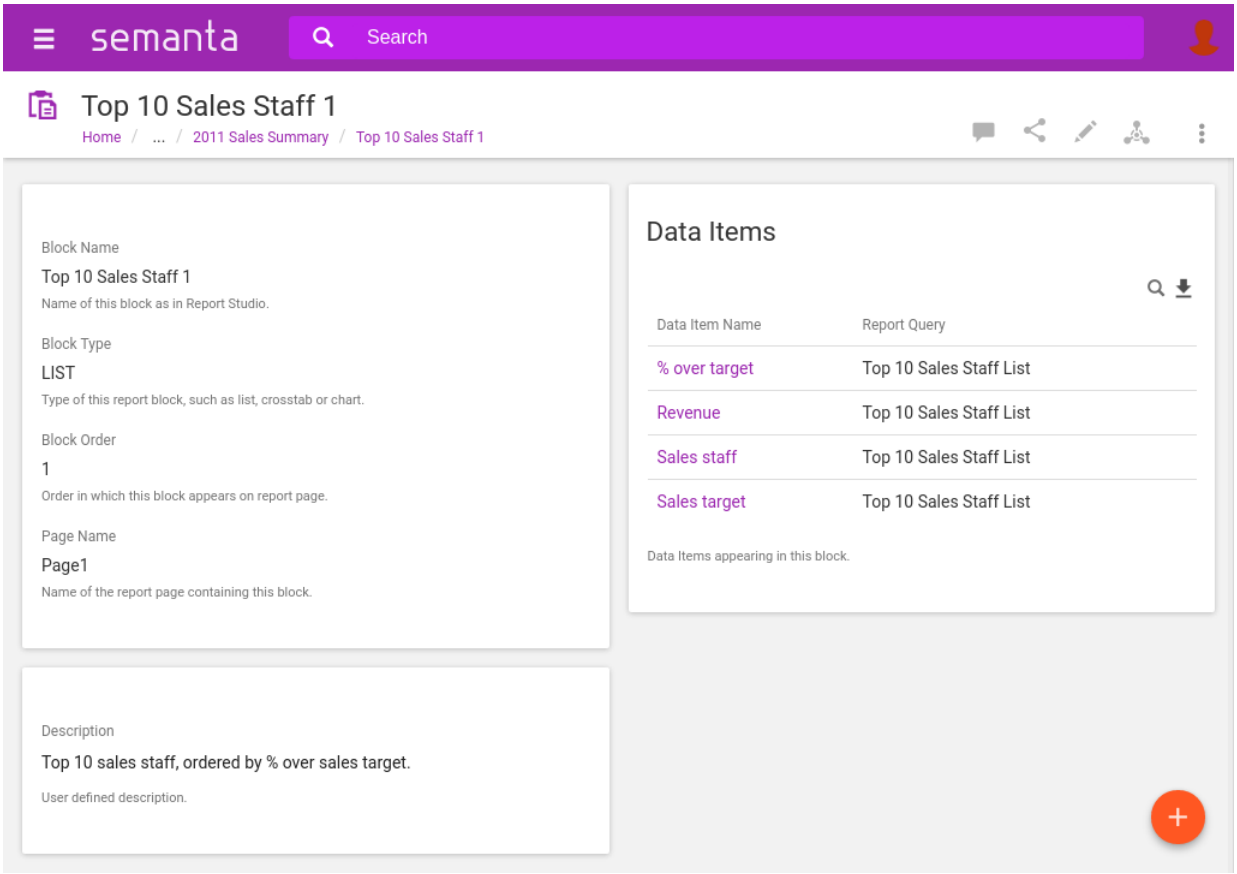
Fragment 4.2: Definice aplikace Schema

```
<xforms-app>
    <xid>coc-schema</xid>
    <entry-name>Schema</entry-name>
    <extended-name>Schema ${entry.name}</extended-name>
    <icon>iVBORw0KGgoAAAANSUhEUgAAAEAAAABACAYAAACqaXHeAA...</icon>
    ...
```

4.3 Aplikace Block

Obrázek 4.3 zobrazuje entry aplikace Block, které reprezentuje blok reportu.

Ve skupině vlevo vidíme název bloku, jeho typ, pořadí na stránce a název stránky, ve které je blok obsažen. Na začátku této kapitoly jsme uvedli, že stránku reportu nechceme reprezentovat jako samostatnou aplikaci. Stránku tedy reprezentujeme jako field na aplikaci Block.



Obrázek 4.3: Entry aplikace Block

Pravá skupina zobrazuje tabulkou seznam datových položek reportu (aplikace Data Item), které tento blok zobrazuje. Vazba na aplikaci Data Item je realizována opět pomocí subaplikačního fieldu.

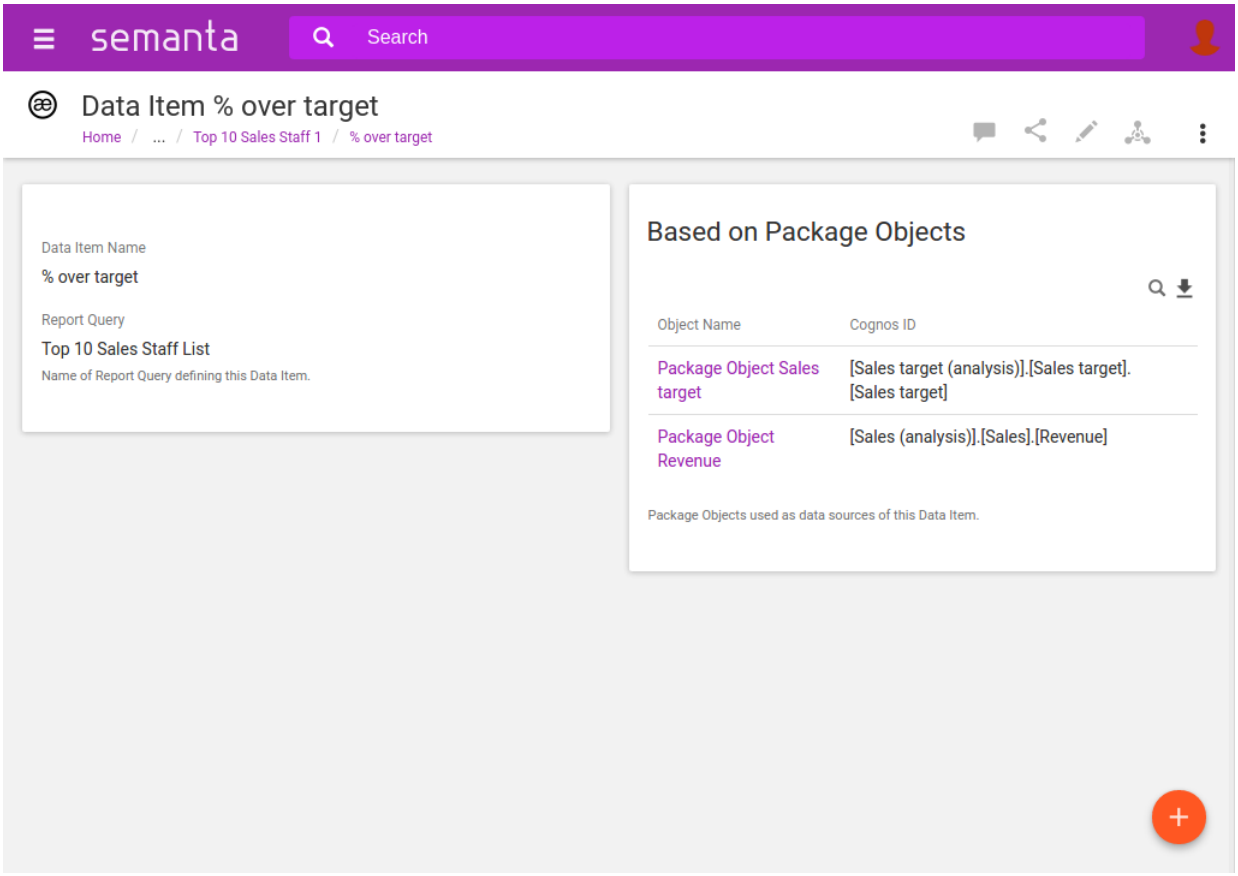
Levá dolní sekce obsahuje uživatelem dodaný popis bloku reportu.

4.4 Aplikace Data Item

Obrázek 4.4 zobrazuje entry aplikace Data Item, které reprezentuje datovou položku reportu.

Skupina vlevo zobrazuje název datové položky a název query, do které v rámci reportu logicky patří. Fieldem obsahující název query realizujeme zjednodušení diskutované v začátku kapitoly, kdy pro query reportu nezavádíme samostatnou aplikaci.

Skupina vpravo obsahuje seznam objektů modelu, které tato datová položka využívá jako zdroje dat. Vazba na objekt modelu je typu reference a je realizována fieldem typu `<relations>`.



Obrázek 4.4: Entry aplikace Data Item

Skupina vlevo dole obsahuje opět pouze field pro uživatelem definovaný popis. Na obrázku není skupina vidět, protože její jediný field nemá v tomto případě vyplněnou hodnotu.

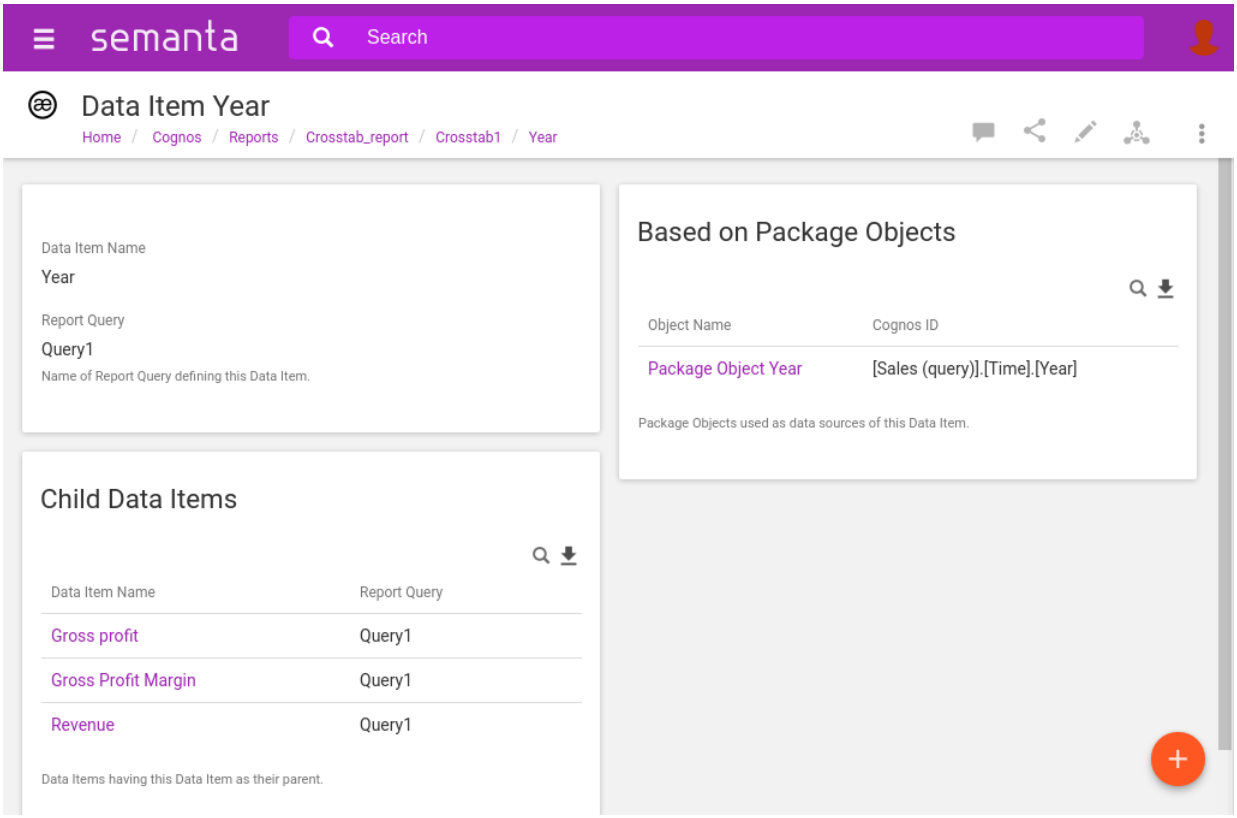
Obrázek 4.5 zobrazuje entry aplikace Data Item pro datovou položku kontingenční tabulky. Tato datová položka má potomky – datové položky. To odpovídá tomu, že v kontingenčních tabulkách mohou datové položky nabývat hierarchií.

4.5 Aplikace Model

Obrázek 4.6 zobrazuje entry aplikace Model.

Skupina vlevo zobrazuje název modelu a cestu k modelu (search path).

Skupina vpravo obsahuje tabulkový výpis balíčků modelu. Jedná se pouze o ty balíčky modelu, které používá jako datový zdroj některý report. Vazba mezi modelem a balíčkem modelu je typu rodič – dítě a je tedy realizována subaplikačním fieldem.



Obrázek 4.5: Entry aplikace Data Item, datová položka s potomky

Skupina vlevo dole obsahuje field s uživatelem definovaným popisem modelu.

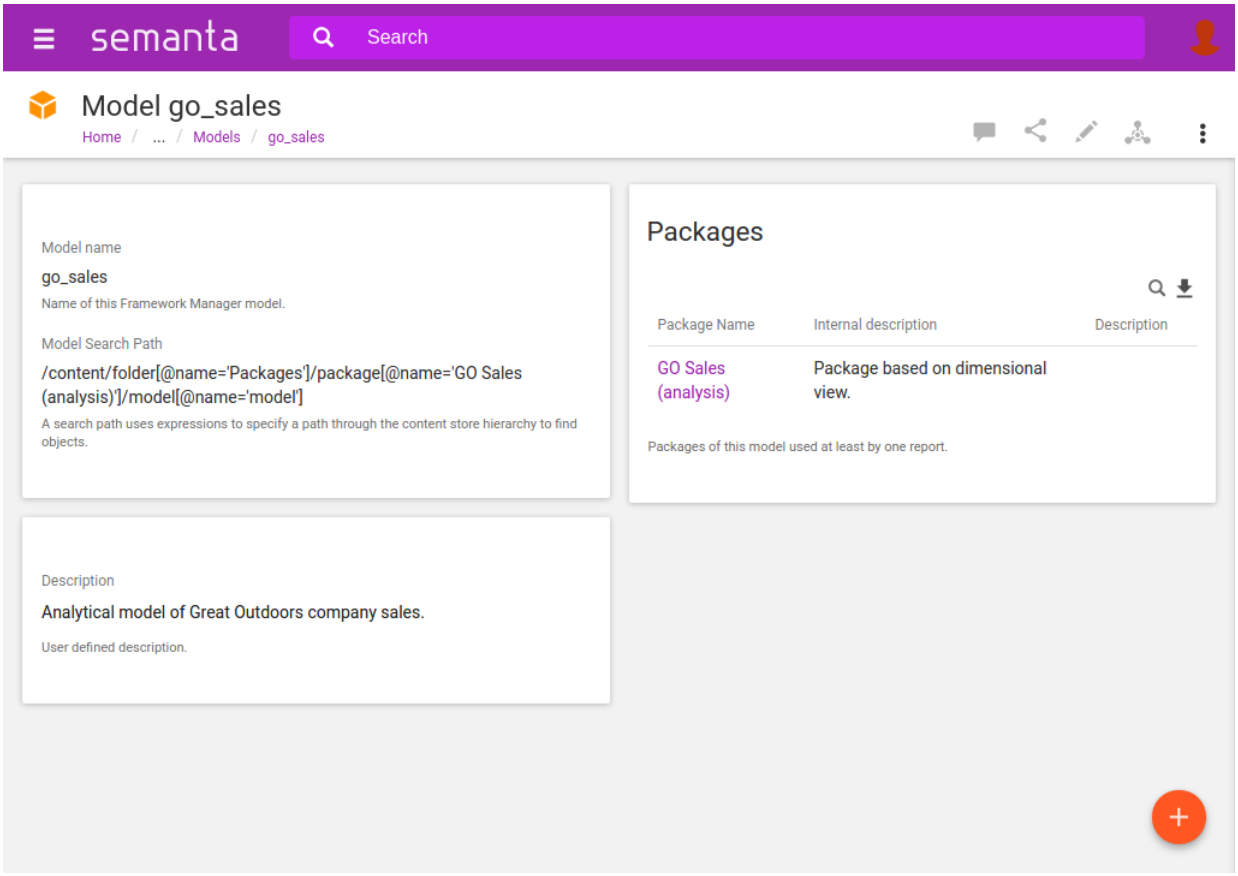
4.6 Aplikace Package

Obrázek 4.7 zobrazuje entry aplikace Package, která reprezentuje balíček modelu.

Skupina vlevo zobrazuje název balíčku a jeho interní popis, který je definován v rámci platformy IBM Cognos.

Skupina vpravo obsahuje tabulkový výpis objektů modelu, patřících do tohoto balíčku. Opět jde o vztah rodič – dítě, realizovaný subaplikačním fieldem.

Skupina vlevo dole obsahuje field s uživatelem definovaným popisem balíčku.



Obrázek 4.6: Entry aplikace Model

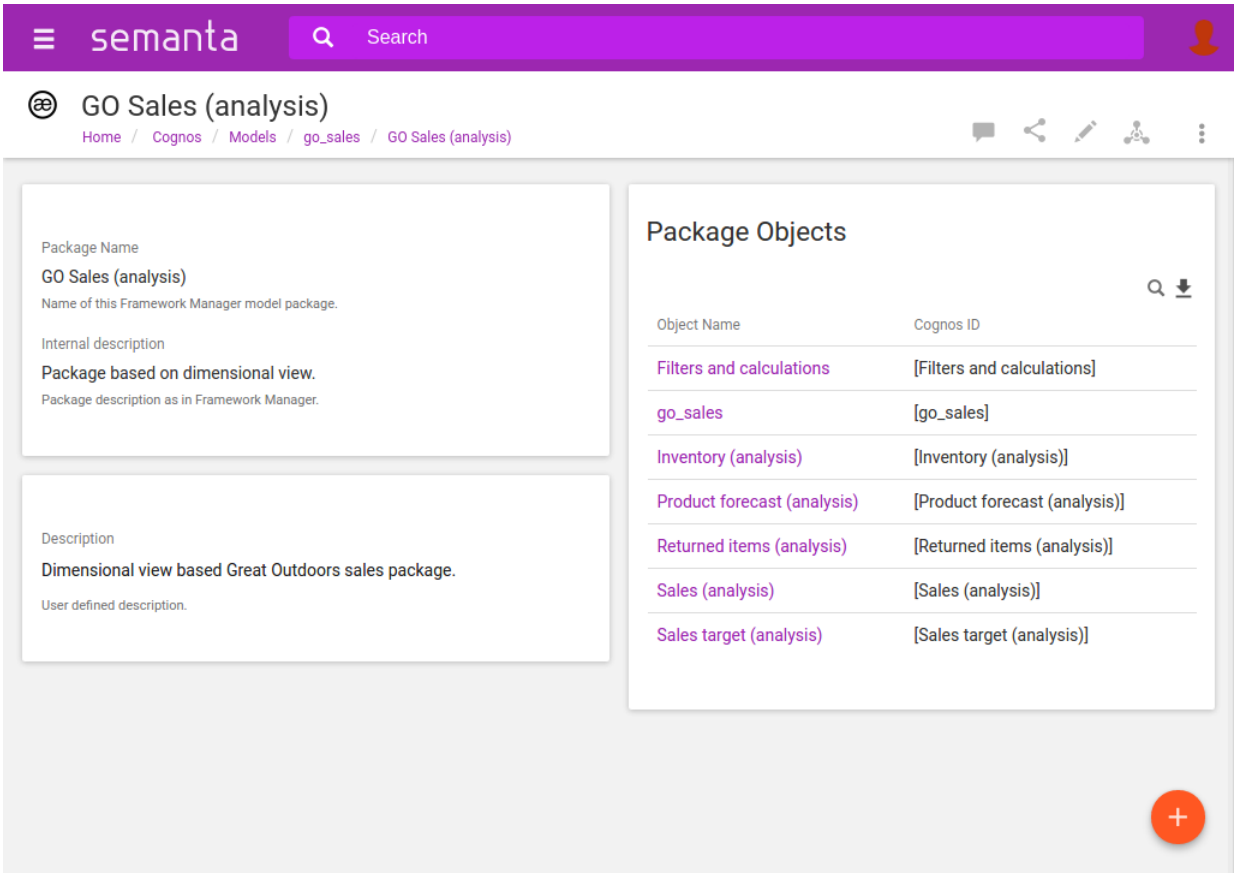
4.7 Aplikace Package Object

Obrázek 4.8 zobrazuje entry aplikace Package Object, která reprezentuje objekt modelu.

Skupina vlevo zobrazuje název objektu a jeho interní identifikátor (objectId) v rámci datového modelu. Dále je zde typ objektu v terminologii platformy Cognos (na obrázku jde o objekt typu míra) a datový typ (na obrázku je číselný datový typ).

Skupina vpravo nahoře obsahuje tabulkový výpis databázových sloupců (aplikace Column), ze kterých tento objekt modelu čerpá data. Vztah je realizován fieldem typu `<relations>`, takže vlastníkem vztahu je aplikace Package Object.

Skupina vpravo dole obsahuje tabulkový výpis datových položek, které využívají tento objekt jako zdroj dat. Jedná se o druhou stranu vazby, definované na aplikaci Field. Vazba je realizována fieldem typu `<lucene-query>` (viz kapitola 2).



Obrázek 4.7: Entry aplikace Package

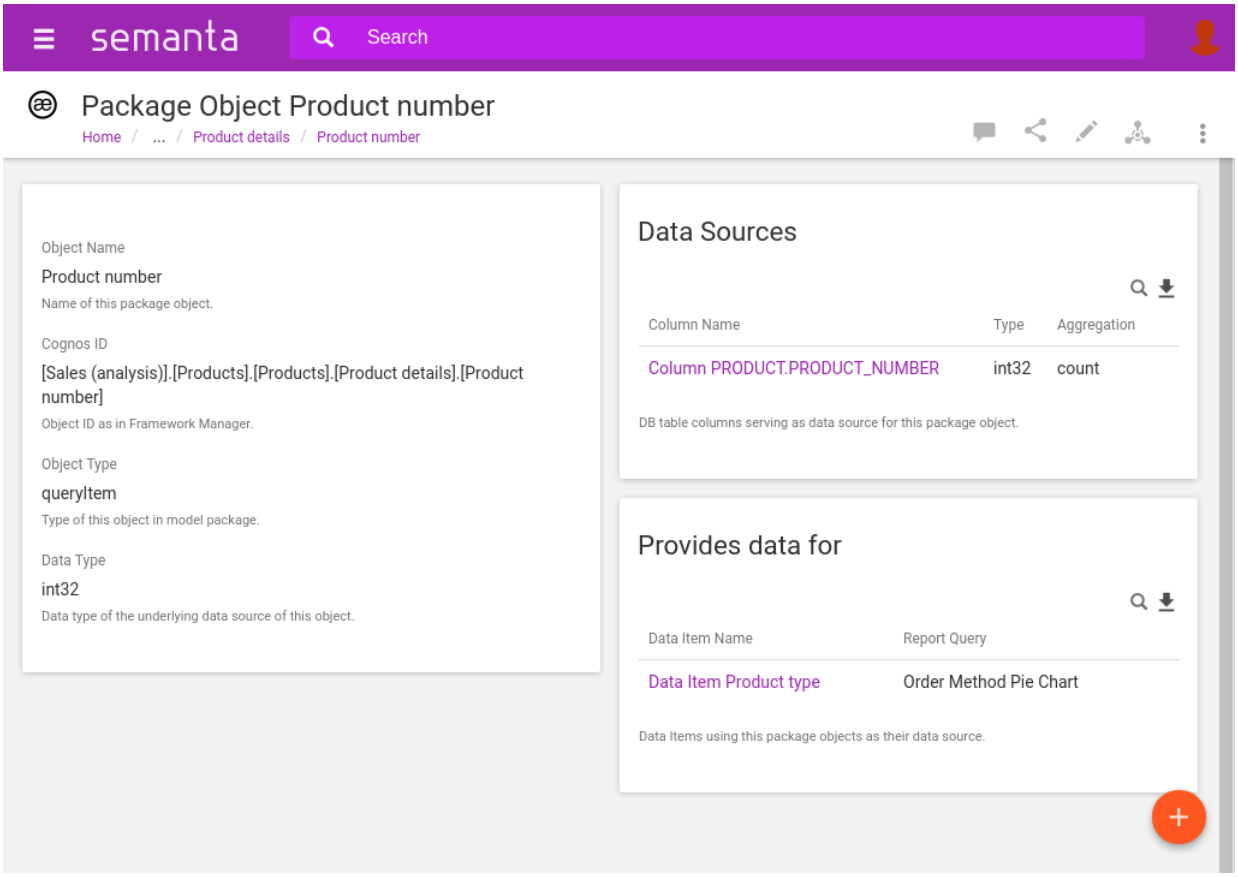
Skupina vlevo dole obsahuje field s uživatelem definovaným popisem (na obrázku je skrytá).

4.8 Aplikace Schema

Obrázek 4.9 zobrazuje entry aplikace Schema, která reprezentuje schéma v relační databázi.

Skupina vlevo zobrazuje název schématu. Skupina vpravo obsahuje tabulkový výpis databázových tabulek, náležících do tohoto schématu. Vztah mezi schématem a tabulkou je realizován subaplikačním fieldem. Na obrázku vidíme, že tabulka defaultně paginuje po 10 položkách.

Skupina vlevo dole obsahuje field s uživatelem definovaným popisem schématu (na obrázku je skrytá).



Obrázek 4.8: Entry aplikace Package Object

4.9 Aplikace Table

Obrázek 4.10 zobrazuje entry aplikace Table, která reprezentuje tabulku v relační databázi.

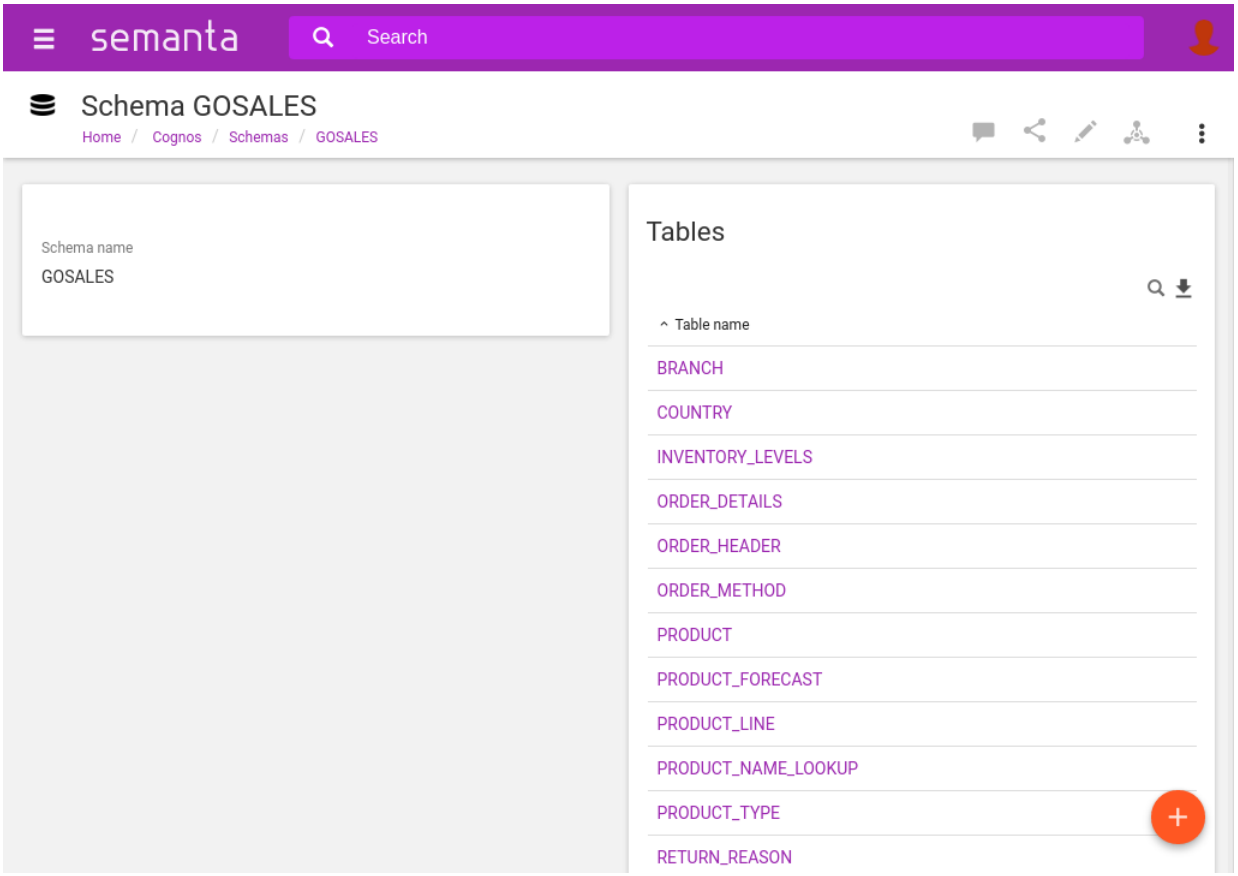
Skupina vlevo zobrazuje název tabulky. Skupina vpravo obsahuje tabulkový výpis sloupců tabulky. Vztah mezi tabulkou a sloupcem je realizován subaplikačním fieldem.

Skupina vlevo dole obsahuje field s uživatelem definovaným popisem tabulky.

4.10 Aplikace Column

Obrázek 4.11 zobrazuje entry aplikace Column, která reprezentuje sloupec v tabulce v relační databázi.

Skupina vlevo zobrazuje název sloupce, jeho datový typ a typ agregace sloupce. Skupina vpravo obsahuje tabulkový výpis objektů modelu, které po-



Obrázek 4.9: Entry aplikace Schema

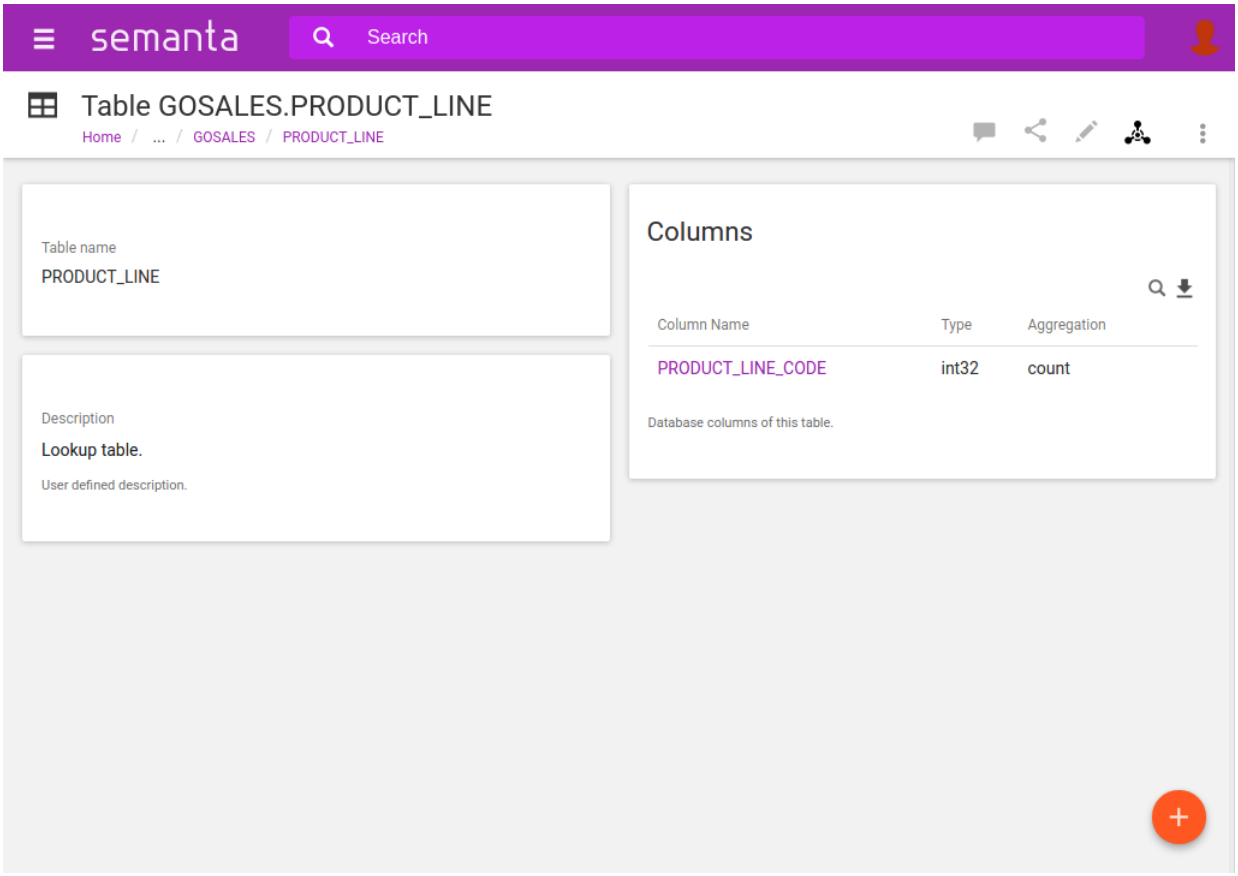
užívají tento sloupec jako zdroj dat. Jedná se o opačný směr vztahu *has data source*, vlastněným aplikací Package Object. Vztah je realizován fieldem typu `<lucene-query>`.

Skupina vlevo dole obsahuje field s uživatelem definovaným popisem sloupce (na obrázku je skrytá).

4.11 Definice nahrávací úlohy

Každé z devíti aplikace odpovídá jedna podúloha, která realizuje vytváření objektů jedné aplikaci z dat jedné nebo více tabulek v relační databázi.

V rámci podúlohy je v předpisu vytvoření entry nutné zadefinovat, jaké entry je rodičem vznikajícího entry. Vazba rodič – dítě se na úrovni nahrávací úlohy definuje pouze v tomto směru. V relačním schématu, které jsme si zadefinovali na konci kapitoly 3 máme dostatek informací v podobě cizích klíčů, abychom byli u každé aplikace schopni dodat informaci o rodiči. Proto na pořadí definice jednotlivých podúloh (které odpovídá pořadí zpracovávání pod-

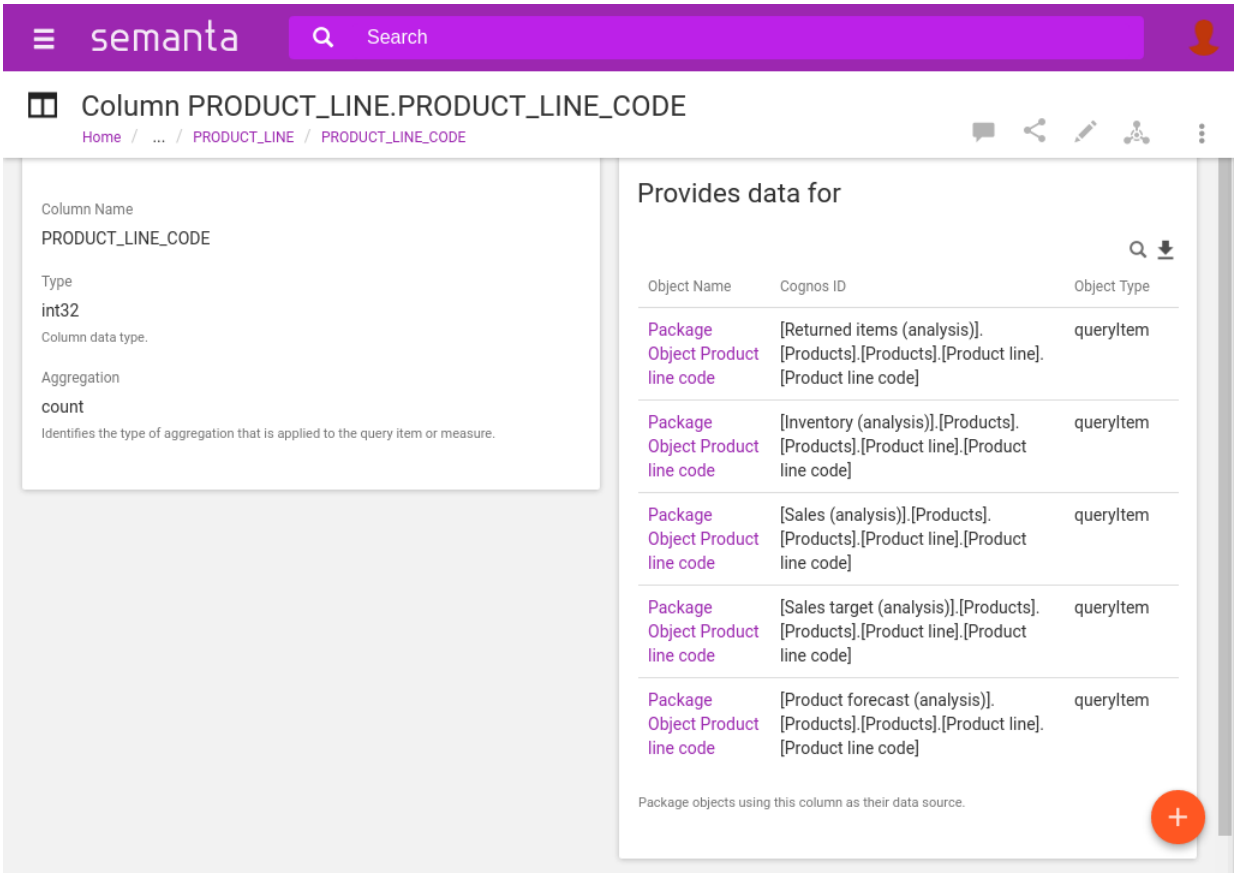


Obrázek 4.10: Entry aplikace Table

úloh v rámci nahrávací úlohy) nezáleží. V rámci zpracování nahrávací úlohy není nutné, aby při vytváření entry aplikace A, které vlastní vazbu na aplikaci B existovalo entry aplikace B.

Speciálním případem jsou aplikace Report, Model a Schema, které nemají aplikace – rodiče. Pro tyto aplikace využijeme jako rodiče aplikaci Folder, dodávanou v rámci platformy XF3. Budeme při definici nahrávací úlohy předpokládat, že existují tři entry typu Folder (například entry Reports, Models a Schemas). Každé ze tří entry typu folder odpovídá jedné kořenové aplikaci a bude mít zadefinovaný *integrační kód*, což je speciální field, který umožní v rámci nahrávací úlohy snadné nalezení entry (viz kapitola 2).

Konceptu nahrávací úlohy jsme se věnovali v kapitole 2, proto rozebereme jenom ty podúlohy, které jsou něčím zajímavé.



Obrázek 4.11: Entry aplikace Column

4.12 Podúloha pro nahrání aplikace Report

Nejsložitější je podúloha pro nahrání aplikace Report, proto si ji rozebereme. Fragment 4.3 ukazuje její definici.

Fragment 4.3: Definice podúlohy pro nahrání aplikace Report

```
<for-each-row>
    <sub-job-id>reports</sub-job-id>
    <of-query>
        <sql>
            <connection-id>ce_mart</connection-id>
            <query-body>
                SELECT
                    r.report_puid as report_id,
                    r.name,
                    r.search_path as searchPath,
                    r.description as internalDescription,
                    r.created as created,
                    r.last_modified as lastModified,
```

```
                    r.enabled,
                    r.visible,
                    r.package_id as report_package_id
                FROM
                    COC_REPORT r
            </query-body>
        </sql>
    </of-query>
    <define-entry>
        <app>coc-report</app>
        <parent>$lu.getEntryForIntegrationCode("coc-reports").getXid()</parent>
        <parent-field>entries</parent-field>
        <xid>${report_id}</xid>
        <name>${name}</name>
        <fields><![CDATA[
            #set($rootUrl=${lu.toXidEncoded($search_path)})
            #set($openableUrl="http://10.0.1.35/ibmcognos/cgi-bin/cognos.cgi?b_action=cognosViewer
            &ui.action=run&ui.object=${lu.encodeCognosSearchPathParam($search_path)}")
            #set($airType="cognos")
            ]]>
        </fields>
        <do-not-touch>
            <field>description</field>
            <field>thumbnail</field>
        </do-not-touch>
        <process-all-rows>
            <of-query>
                <sql>
                    <connection-id>ce_mart</connection-id>
                    <query-body>
                        SELECT
                            p.package_id
                        FROM
                            COC_PACKAGE p
                        WHERE
                            p.package_id = '${report_package_id}'
                    </query-body>
                </sql>
            </of-query>
            <set-fields>
                #set($packagesUsed=$lu.targetsFromRows($allRows, "package_id"))
            </set-fields>
        </process-all-rows>
    </define-entry>
</for-each-row>
```

V elementu `<query-body>` definujeme tělo primárního SQL dotazu. Protože při přiřazování hodnot fieldům vznikajícího entry se automaticky párují názvy fieldů s kontextem podúlohy, která obsahuje výsledek primárního a případného sekundárního SQL dotazu (viz kapitola 2), můžeme si ušetřit práci vhodným pojmenováním sloupců výsledků dotazu, aby odpovídaly názvům příslušných fieldů aplikace. Například sloupec `last_modified` pojmenujeme ve výsledku `lastModified`, což odpovídá názvu fieldu.

Nahrávací úloha používá standardní datové typy rozhraní JDBC, proto se nemusíme starat o převody u běžných datových typů, jako je například datum v případě sloupce `last_modified`.

Pro získání rodiče využíváme již zmíněný integrační kód, předpokládáme existenci entry typu Folder, které bude obsahovat všechny entry typu Report.

V elementu `<fields>` definujeme pouze ty fieldy, které vyžadují dodatečnou transformaci. Jsou to fieldy `rootUrl`, `openableUrl` a `airType`. Všechny tyto tři fieldy jsou technické fieldy, kterými

Element `<do-not-touch>` umožňuje nadefinovat, které fieldy má nahrávací proces ignorovat. V našem případě takto označíme fieldy `description` a `thumbnail`, které vyplňuje manuálně uživatel. Nechceme, by při nahrání nové verze reportu byly smazány informace, které uživatel do těchto fieldů vyplnil.

V sekundárním dotazu (definovaném v rámci elementu `<process-all-rows>`) naplníme field `packagesUsed`, který definuje vztah `uses-package` s aplikací Package. Data pro naplnění tohoto fieldu získáme z tabulky `COC_PACKAGE`, přičemž použijeme hodnotu cizího klíče (`report_package_id`), kterou máme k dispozici z primárního dotazu. Transformaci hodnot sloupce s primárním klíčem (sloupec `package_id` tabulky `COC_PACKAGE`) na objekt relace provedeme stejným způsobem, jaký jsme popsali v kapitole 2.

4.13 Podúloha pro nahrání aplikace Package Object

V sekci 3.7.4 jsme zmínili, že vzájemnou závislost objektů datového modelu vyřešíme na úrovni nahrávací úlohy. Fragment 4.4 zobrazuje část podúlohy, která se týká určení rodiče vznikajícího entry.

Element `<parent>` obsahuje Velocity výraz, který provede rozklad identifikátoru *objectId* objektu modelu. Ten je ve tvaru `[Sales (query)].[Time].[Year]` – tečky oddělují jednotlivé názvy objektů, poslední název je název aktuálního objektu. V tomto příkladu je objekt `Sales (query)` rodičem objektu `Time`, který je rodičem objektu `Year`.

Použitý Velocity výraz získá podřetězec od začátku identifikátoru až po poslední tečku. Tento podřetězec odpovídá názvu rodičovského objektu aktuálního objektu.

Takto získaný název rodiče převedeme na identifikátor, který používáme pro entry objektů datového modelu. Vidíme v elementu `<xid>`, že identifikátor tohoto typu entry vzniká zakódováním názvu entry pomocí metody `toXidEncoded`, dostupné na pomocném objektu `LoadingUtils`. Nic nám nebrání toto zopakovat při specifikování rodiče, název rodiče odpovídá již zmíněnému podřetězci.

Zpracování takto definované podúlohy překvapivě nevyžaduje, aby rodičovské entry existovaly před nahráváním jejich dětí. XF3 dokáže zpracovat

entry ve vztahu rodič – dítě, i když se v rámci nahrávání objeví ve špatném pořadí.

Fragment 4.4: Definice rodiče pro podúlohu aplikace Package Obejct

```
<define-entry>
    <app>coc-package-object</app>
    <parent>#if(${cognos_id.contains('.')})${lu.toXidEncoded(${cognos_id.substring(0, ${cognos_id.lastI
    <parent-field>objects</parent-field>
    <xid>$lu.toXidEncoded(${cognos_id})</xid>|
    ...
```

4.14 Podúloha pro nahrání aplikace Block

Tato podúloha je pro primární SQL dotaz používá tři tabulky schématu – `COC_LAYOUT`, `COC_PAGE` a `COC_BLOCK` (viz sekce 4.1). Fragment 4.5 ukazuje definici primárního SQL dotazu, kde dochází ke spojení tří zmíněných tabulek.

Fragment 4.5: Definice primárního dotazu pro podúlohu aplikace Block

```
<query-body>
    SELECT
        la.report_id,
        la.layout_id,
        pa.page_id,
        pa.name as page_name,
        bl.block_id,
        bl.name as block_name,
        bl.type,
        bl.block_order
    FROM
        COC_LAYOUT la
    JOIN
        COC_PAGE pa on (pa.layout_id = la.layout_id)
    JOIN
        COC_BLOCK bl on (bl.page_id = pa.page_id)
</query-body>
```

4.15 Konfigurace komponenty Semanta Air

V kapitole 2 jsme naznačili, jak funguje komponenta Air. Ta dostává požadavky s URL zdroje (například URL reportu v platformě Cognos) a typem zdroje, pokud byl detekován (například typ Cognos). Následně URL transformuje na takzvanou *root URL*, kterou se pokusí vyhledat mezi entry aplikace, které implementují roli *Url Catalog* (technické označení pro Air). V našem případě bude roli Url Catalog implementovat aplikace Report. Každému reportu v platformě Cognos odpovídá jednoznačná URL, což je podmínka pro to, aby Air mohl nad touto platformou fungovat.

Aplikace označuje, že je součástí Airu elementem `<role>urlcatalog</role>` v sekci `<implements>`. Dále potřeba, aby na aplikaci byly přítomny technické fieldy s kódy `rootUrl`, `openableUrl` a `airType`. Do těchto fieldů bude Air ukládat technické informace o entry. Field `airType` slouží k uložení typu zdroje, jak byl detekován Airem náběrem stránky. Typ zdroje následně určuje, jaké transformace se mají aplikovat na URL. Field `openableUrl` slouží k uložení původní URL zdroje. Field `rootUrl` obsahuje stejnou URL po transformaci.

Detekci typu `airType` pro platformu Cognos implementujeme triviálním způsobem. Pokud se v URL vyskytuje řetězec „cognos“, nastavíme jako typ zdroje hodnotu `cognos`. Detekce typu se implementuje v rámci JavaScriptové části Airu.

Transformaci URL pro typ `cognos` postavíme na faktu, že URL reportů v aplikaci Report Viewer platformy Cognos obsahují parametr, ve kterém je obsažena celá search path reportu. Každý report má v rámci Content Store definovanou unikátní search path, proto nám stačí jako `rootUrl` uložit hodnotu search path. Hodnotu ještě zakódujeme pomocnou metodou API XF3 způsobem, který umožní uložení její hodnoty do Apache Lucene bez ohledu na možné problematické znaky.

Chceme dále dosáhnout stavu, kdy po loadu reportu bude tento okamžitě přístupný komponentou Air na příslušné URL v Report Vieweru. Toho dosáhneme tak, že v rámci nahrávací podúlohy pro aplikaci Report nastavíme stejné hodnoty technických fieldů Airu, jako kdyby došlo k jejich vyplnění přímo komponentou Air. Odpovídající část nahrávací úlohy je vidět ve fragmentu 4.3.

Field `airType` triviálně vyplníme hodnotou `cognos`. Field `rootUrl` vyplníme zakódovanou hodnotou search path reportu, kterou máme o každém reportu uloženou v databázi. Tím dosáhneme stejného výsledku jako transformátor URL typu Cognos popsaný výše.

Field `openableUrl` potřebujeme naplnit hodnotou odpovídající URL reportu v Report Vieweru, aby bylo možné z entry reportu v XF3 přejít přímo na report v platformě IBM Cognos. URL tedy nemusí být identická, jako by ji vygenerovala aplikace Report Viewer, ale musí ji platforma Cognos zpracovat tak, aby došlo k otevření správného reportu. Experimentálním způsobem jsme zjistili parametry. které musí být přítomny (viz řádek `#set($openableUrl...` fragmentu 4.3).

4.16 Integrace Semanta Air do platformy IBM Cognos

Obohacení cílové stránky Airem je realizováno spuštěním kódu v jazyce JavaScript, který vytvoří na stránce element iframe a vloží do něj relevantní obsah z XF3. Spuštění tohoto kódu je možné přes obecný mechanismus spuštění,

který Air nabízí a který je založen na bookmarkletu (velmi krátká JavaScriptová aplikace, která je uložena jako záložka v prohlížeči uživatele). Tento způsob vyžaduje, aby uživatel po otevření reportu ručně aktivoval Air otevřením záložky s bookmarkletem. Při obnovení stránky (například při přechodu na jiný report) je nutné proces spuštění zopakovat.

Aby došlo ke spuštění Airu automaticky při otevření reportu v Report Vieweru, bylo nutné spouštěcí kód v JavaScriptu zaintegrovat do platformy Cognos BI. Povedlo se nám v instalaci platformy lokalizovat kód v JavaScriptu, který je spouštěn na každé stránce reportu ve webové aplikaci Report Viewer. Do tohoto kódu se nám podařilo vložit vlastní kód pro automatické spuštění Airu nad jakýmkoliv reportem.

Uživatelský zážitek z integrace komponenty Air s platformou IBM Cognos následně působí zcela bezešvým dojmem. Ukázka integrace je na obrázku A.2

4.17 Vyhledávání nad vytvořenými daty

Obrázek A.5 ukazuje, jak funguje v platformě XF3 rychlé vyhledávání nad entries jednotlivých aplikací.

Obrázek A.6 zobrazuje funkci pokročilého vyhledávání. Vidíme, že do výsledku na dotaz „gross profit“ se dostalo i entry reportu. To je způsobeno tím, že vyhledávání je fulltextové a entry reportu obsahuje hledaný výraz ve fieldu s interním popisem.

Závěr

Cílem práce bylo vytvořit katalog reportů na platformě XF3, který umožní katalogizovat reporty z IBM Cognos. Katalog měl umožňovat snadné vyhledávání reportů, jejich částí a jejich datových zdrojů. K primárnímu plnění katalogu mělo docházet automaticky, přenosem informace přímo z IBM Cognos. Na tuto kostru však muselo být možné doplňovat další údaje manuálně. Dalším požadavkem byla dostupnost takto vzniklé dokumentace přímo z reportů Cognosu.

Naplnění cílů práce

Můžeme konstatovat, že všech cílů práce bylo rámcově dosaženo, viz diskuze výsledné aplikace v kapitole 4. Byl naimplementován loader metadat reportů IBM Cognos, který ukládá data relační databáze. Dále byly vytvořeny nahrávací úlohy a aplikace v platformě XF3, které tato data zobrazují. Platforma XF3 nad těmito daty umožnuje vyhledávání a vizualizaci vztahů jednotlivých entit. Je možné manuálně doplňovat informace k automaticky nahraným reportům. Dostupnost dokumentace v platformě IBM Cognos je zajištěna pomocí komponenty Semanta Air.

Pochopitelně, na každé z komponent řešení by bylo možné dále pracovat. Některé konkrétní možnosti rozšíření diskutujeme v následujících sekci. Mimo to je třeba další práce k uvedení řešení do stavu vhodného pro komerční využití.

Tyto další práce zahrnují detailní otestování nad jednotlivými verzemi platformy IBM Cognos. Dále se jedná o testování zpracování reportů, které využívají méně běžné obraty (například vzorce definované přímo na úrovni datových položek reportu a ne na úrovni logického modelu). Platí, že v podnikové praxi často vznikají reporty fungující, nicméně s nižší kvalitou zpracování než demonstrační reporty dodávané s platformou. Tvůrci reportů v praxi například často umístí text a objekt reportu graficky vedle sebe, ale na úrovni definice reportu je nepropojí. Systém nemá informaci, že popisek patří k tomuto objektu, zatímco uživateli je to z grafické formy zřejmé. Toto jsou problémy, se kterými by se komerční produkt měl nějakým způsobem vypořádat.

Možnosti dalšího rozšíření

Zpracování report queries

V sekci 3.11 jsme popsali, že část XML specifikace reportu, která se týká report queries, pro relativně vyšší komplexitu XML schématu nezpracováváme. Tím přicházíme o poměrně zajímavou část informace o datových položkách reportu. Pokud by se podařilo parsovat alespoň většinu XML elementů, odpovídající jednotlivých typům datových položek, měli bychom v katalogu reportů další užitečnou informaci.

Indexy v XML databázi

V implementaci konektoru je pro běh XQuery kódu použita databáze eXist-db v tzv. embedded režimu, kdy je s aplikací distribuován JAR soubor s databází eXist a ta je následně spuštěna v rámci loaderu bez nutnosti ji instalovat. Při vývoji byla využívána standalone instalace databáze a bylo zjištěno, že nadefinováním vhodných indexů se násobně zrychlí zpracování XML dokumentu data lineage objektů datového modelu. Konfigurace indexů se provádí nahráním konfiguračního XML souboru do databáze eXist. Pokud bychom umožnili nahrávání konfiguračního souboru při inicializaci embedded databáze, došlo by k výraznému zrychlení loaderu.

Vlastní implementace služby pro volání Metadata Service

V sekci 3.7.2 jsme zmínili, že pro komunikaci se službou API Metadata Service je použit vzorový kód IBM (viz [6]). Bylo by vhodné odladit vlastní implementaci této služby, která se ukázala nespolehlivá kvůli relativně vysoké náročnosti obsluhy této části API IBM Cognos.

Literatura

[1] Siegel, E.; Retter, A.: *eXist: A NoSQL Document Database and Application Platform*. O'Reilly Media, 2014, ISBN 9781449337100. Dostupné z: https://library.oreilly.com/book/0636920026525/exist/

[2] IBM: *IBM Cognos Software Development Kit, Version 10.2.2: Software Development Kit Developer Guide*. 2012. Dostupné z: http://public.dhe.ibm.com/software/data/cognos/documentation/docs/en/10.2.2/dg_sdk.pdf

[3] Maxia, G.: datacharmer/test_db: A sample MySQL database with an integrated test suite, used to test your applications and database servers [online]. [cit. 2017-02-15]. Dostupné z: https://github.com/datacharmer/test_db

[4] Microsoft: MSXML SDK Overview: Sample XML File (books.xml) [online]. [cit. 2017-01-24]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms762271.aspx

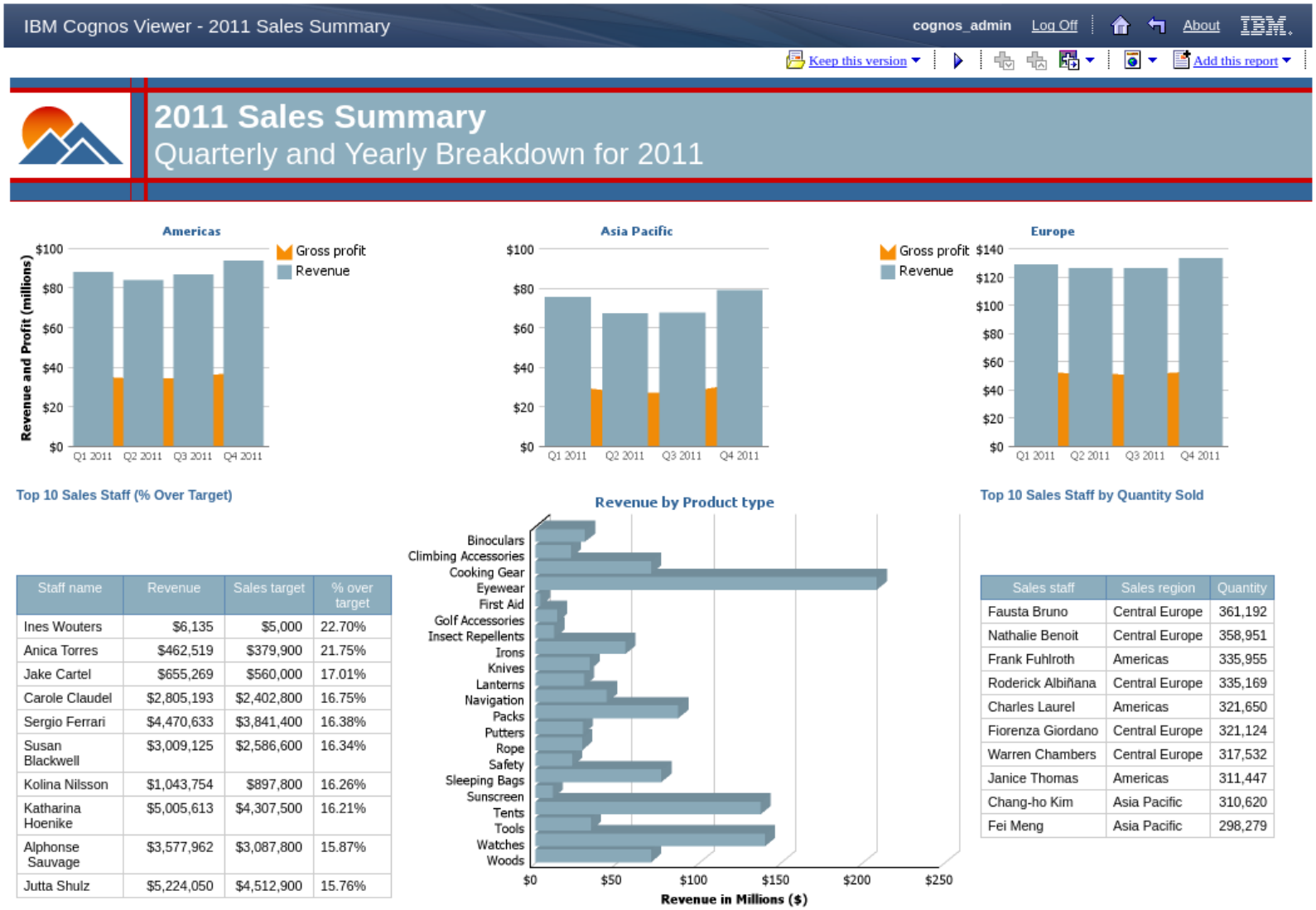
[5] *Scrapy 1.0 documentation [online]*. [cit. 2017-01-24]. Dostupné z: https://doc.scrapy.org/en/1.0/topics/selectors.html

[6] IBM: SDK Sample: How to Run a Lineage Specification [online]. [cit. 2017-02-15]. Dostupné z: http://www-01.ibm.com/support/docview.wss?uid=swg21363958

[7] Walmsley, P.: *XQuery*. O'Reilly, 2006.

Příloha A

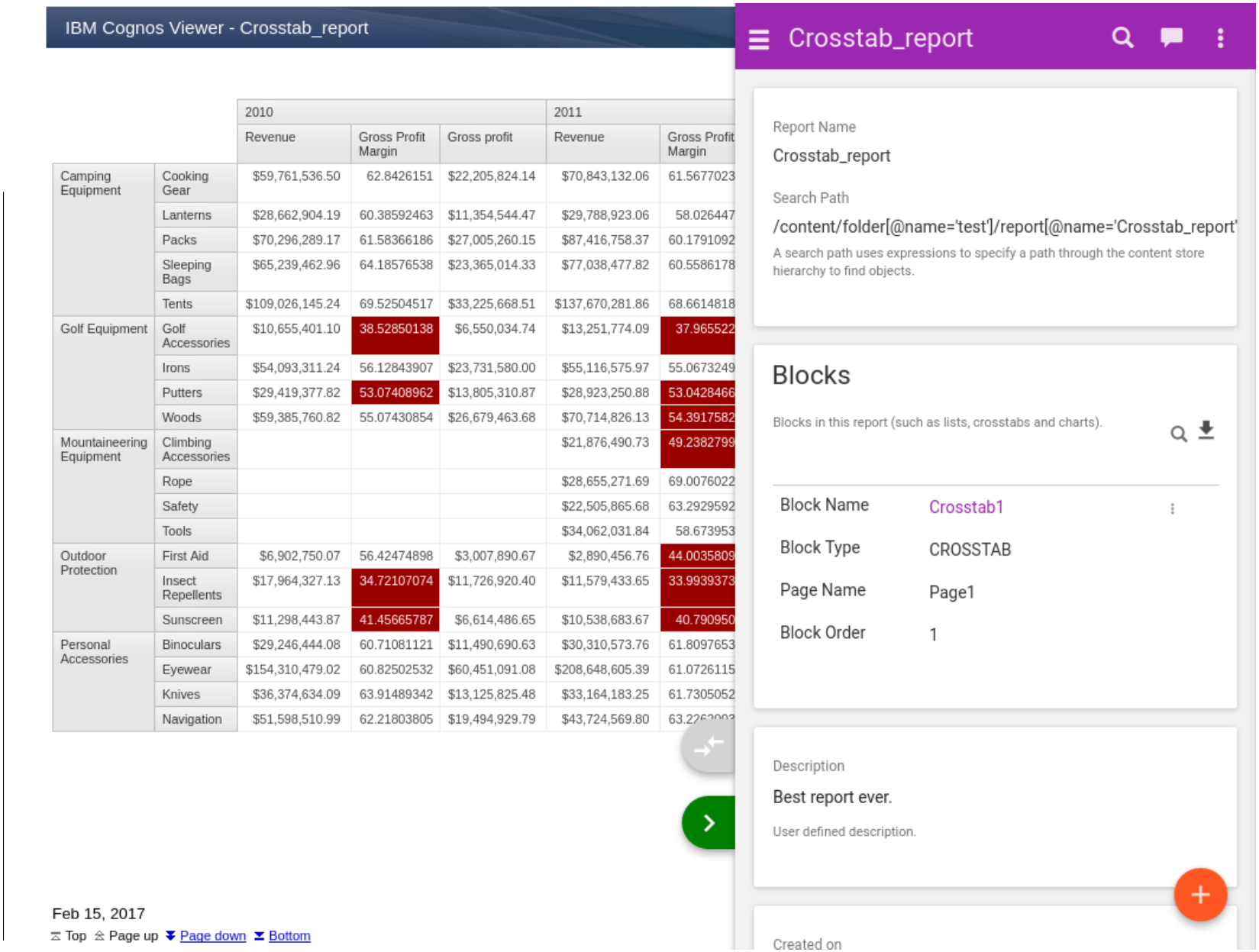
Obrázkové přílohy



Staff name	Revenue	Sales target	% over target
Ines Wouters	$6,135	$5,000	22.70%
Anica Torres	$462,519	$379,900	21.75%
Jake Cartel	$655,269	$560,000	17.01%
Carole Claudel	$2,805,193	$2,402,800	16.75%
Sergio Ferrari	$4,470,633	$3,841,400	16.38%
Susan Blackwell	$3,009,125	$2,586,600	16.34%
Kolina Nilsson	$1,043,754	$897,800	16.26%
Katharina Hoenike	$5,005,613	$4,307,500	16.21%
Alphonse Sauvage	$3,577,962	$3,087,800	15.87%
Jutta Shulz	$5,224,050	$4,512,900	15.76%

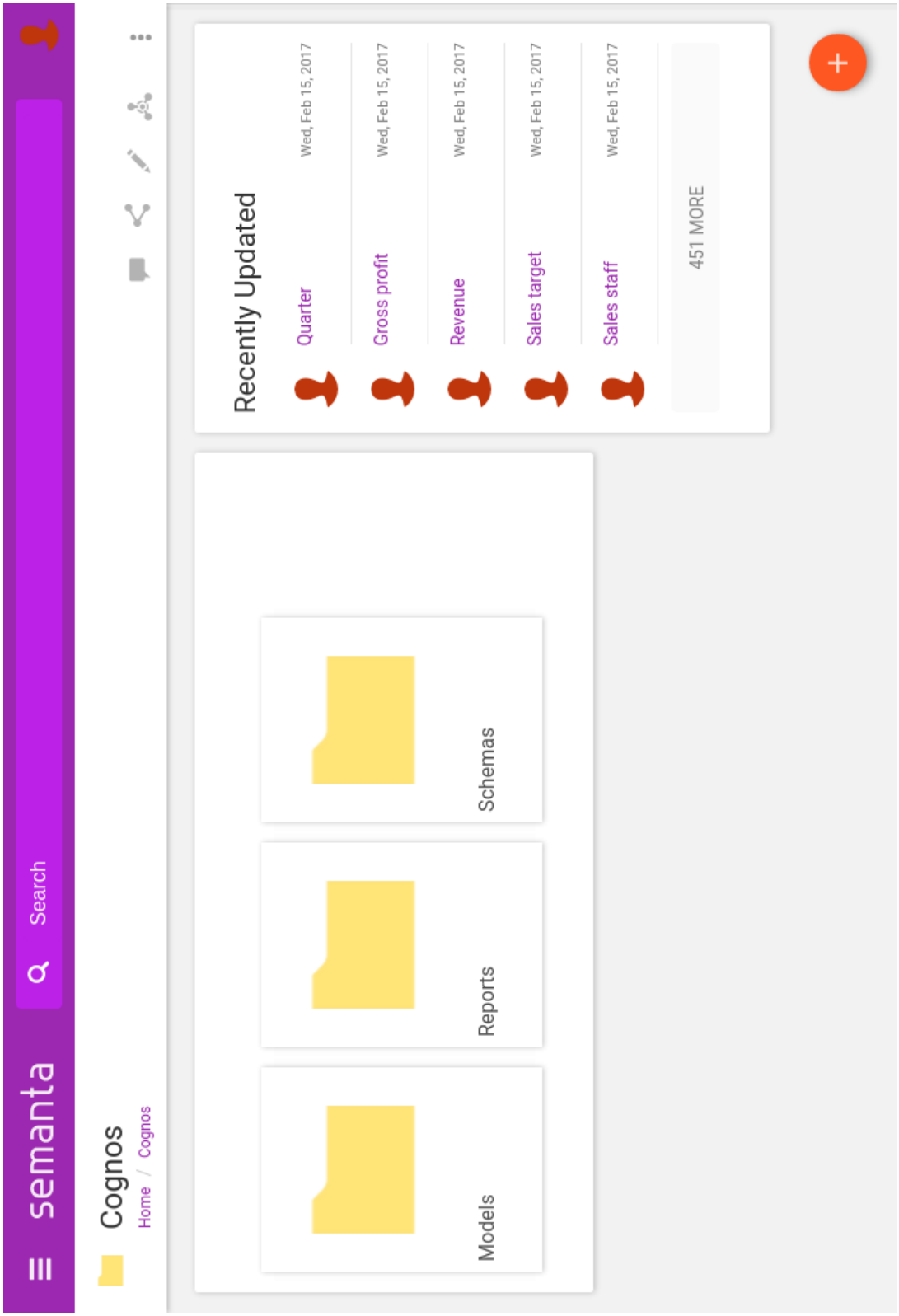
Sales staff	Sales region	Quantity
Fausta Bruno	Central Europe	361,192
Nathalie Benoit	Central Europe	358,951
Frank Fuhlroth	Americas	335,955
Roderick Albiñana	Central Europe	335,169
Charles Laurel	Americas	321,650
Fiorenza Giordano	Central Europe	321,124
Warren Chambers	Central Europe	317,532
Janice Thomas	Americas	311,447
Chang-ho Kim	Asia Pacific	310,620
Fei Meng	Asia Pacific	298,279

Obrázek A.1: Demonstrační report 2011 Sales Summary

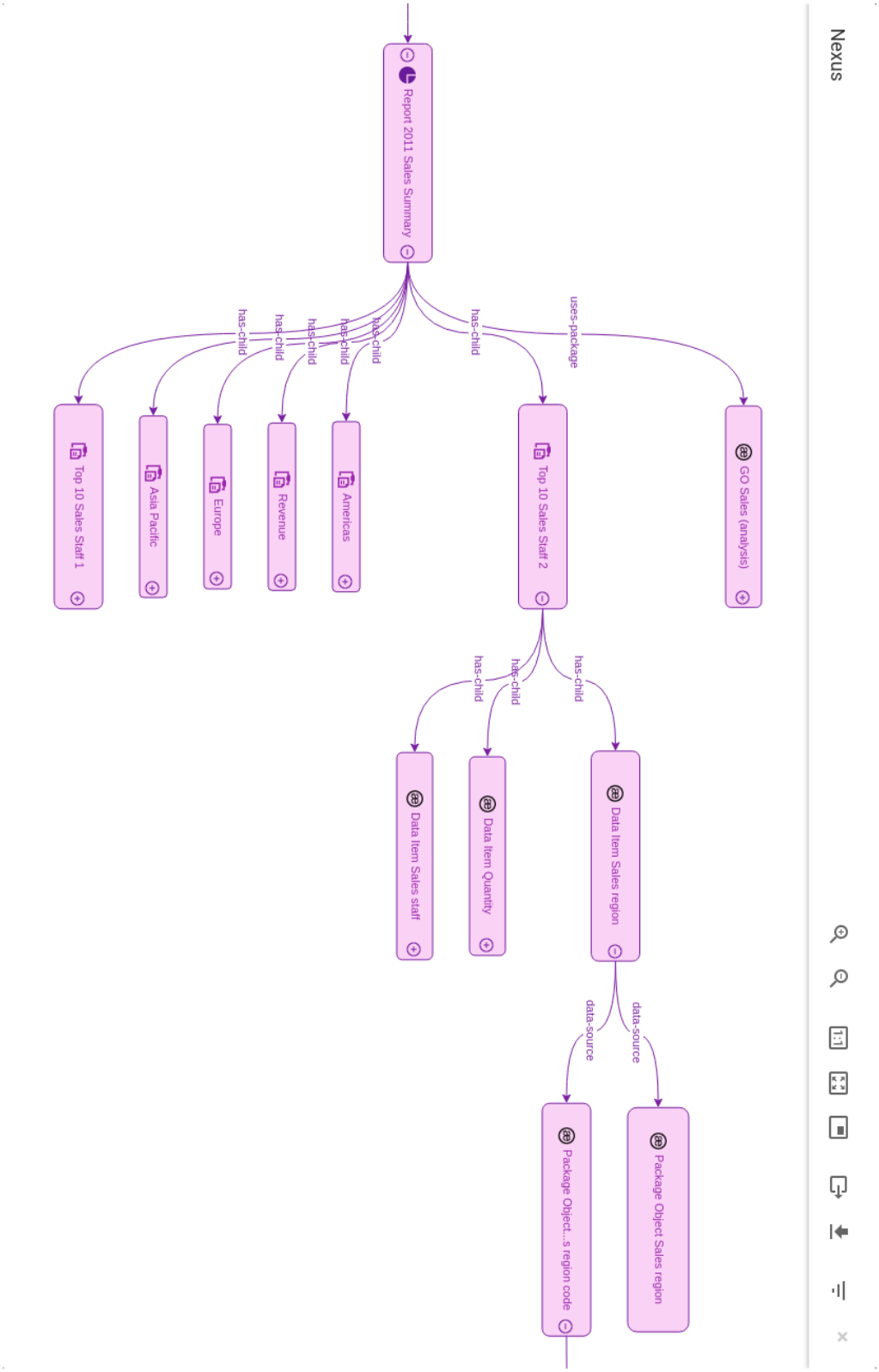


		2010			2011	
		Revenue	Gross Profit Margin	Gross profit	Revenue	Gross Profit Margin
Camping Equipment	Cooking Gear	$59,761,536.50	62.8426151	$22,205,824.14	$70,843,132.06	61.5677023
	Lanterns	$28,662,904.19	60.38592463	$11,354,544.47	$29,788,923.06	58.026447
	Packs	$70,296,289.17	61.58366186	$27,005,260.15	$87,416,758.37	60.1791092
	Sleeping Bags	$65,239,462.96	64.18576538	$23,365,014.33	$77,038,477.82	60.5586178
	Tents	$109,026,145.24	69.52504517	$33,225,668.51	$137,670,281.86	68.6614818
Golf Equipment	Golf Accessories	$10,655,401.10	38.52850138	$6,550,034.74	$13,251,774.09	37.965522
	Irons	$54,093,311.24	56.12843907	$23,731,580.00	$55,116,575.97	55.0673249
	Putters	$29,419,377.82	53.07408962	$13,805,310.87	$28,923,250.88	53.0428466
	Woods	$59,385,760.82	55.07430854	$26,679,463.68	$70,714,826.13	54.3917582
Mountaineering Equipment	Climbing Accessories				$21,876,490.73	49.2382799
	Rope				$28,655,271.69	69.0076022
	Safety				$22,505,865.68	63.2929592
	Tools				$34,062,031.84	58.673953
Outdoor Protection	First Aid	$6,902,750.07	56.42474898	$3,007,890.67	$2,890,456.76	44.0035809
	Insect Repellents	$17,964,327.13	34.72107074	$11,726,920.40	$11,579,433.65	33.9939373
	Sunscreen	$11,298,443.87	41.45665787	$6,614,486.65	$10,538,683.67	40.790950
Personal Accessories	Binoculars	$29,246,444.08	60.71081121	$11,490,690.63	$30,310,573.76	61.8097653
	Eyewear	$154,310,479.02	60.82502532	$60,451,091.08	$208,648,605.39	61.0726115
	Knives	$36,374,634.09	63.91489342	$13,125,825.48	$33,164,183.25	61.7305052
	Navigation	$51,598,510.99	62.21803805	$19,494,929.79	$43,724,569.80	63.2263003

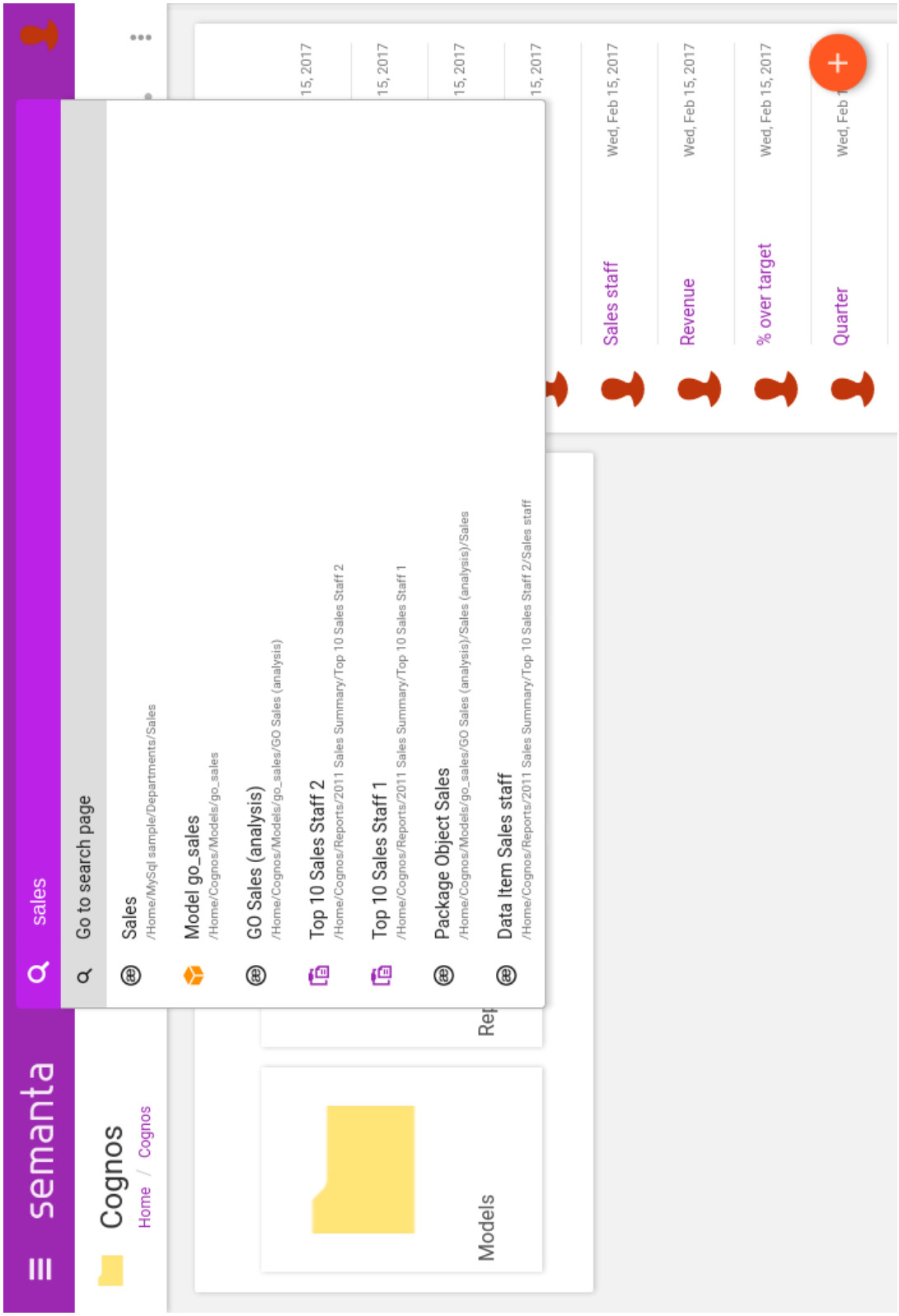
Obrázek A.2: Integrace Semanta Air do Cognos Report Viewer



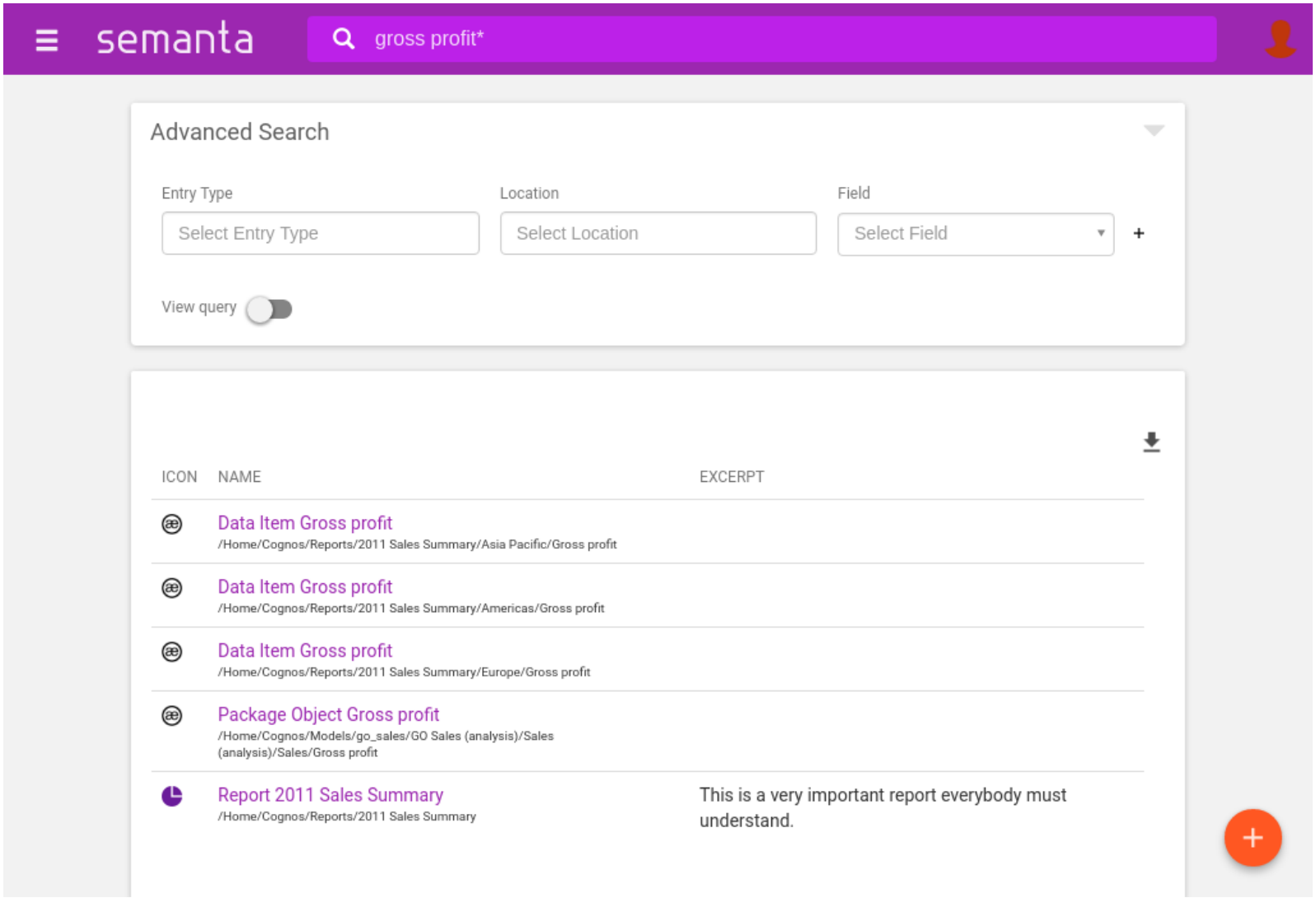
Obrázek A.3: Ukázka platformy XF3



Obrázek A.4: Vizualizace vztahů mezi entries komponentou Nexus



Obrázek A.5: Vyhledávání nad daty aplikací



Obrázek A.6: Pokročilé vyhledávání nad daty aplikací

Příloha B

Seznam použitých zkratek

BI Business intelligence

API Application programming interface

XML Extensible Markup Language

OLAP Online analytical processing

LDAP Lightweight Directory Access Protocol

SDK Software Development Kit

SOAP Simple Object Access Protocol

XSD XML Schema Definition

IDE Integrated Development Environment

JAR Java ARchive

Příloha C

Obsah přiloženého CD

- `diagrams` ..... zdrojový kód UML diagramů
- `text` ..... zdrojové soubory práce
- `sample-apps` ..... vzorové aplikace platformy XF3
- `dp.pdf` ..... text práce ve formátu PDF