Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Framework Spring Data pro řešení přístupu k datům v Java Enterprise aplikacích

Diplomová práce

Autor: Lukáš Duspiva

Studijní obor: Informační management

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové listopad 2014

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne Lukáš Duspiva

Poděkování

Rád bych poděkoval vedoucímu této práce doc. Ing. Filipu Malému, Ph.D. za jeho rady a připomínky. Dále bych rád poděkoval své přítelkyni Kristýně, své rodině a všem, kdo mě podporovali a motivovali během tvorby této práce.

Anotace

Cílem této diplomové práce je přiblížit čtenáři tvorbu aplikační vrstvy pro přístup k datům pomocí frameworku Spring Data. Na úvod jsou představena teoretická východiska, která pomohou lépe pochopit řešenou problematiku. Následně je rozebrána vrstva pro přístup k datům a jsou nastíněny její základní možnosti implementace prostřednictvím známých návrhových vzorů. Hlavní část se pak zabývá popisem frameworku Spring Data, a to především jeho obecnou částí označovanou jako Spring Data Commons. Na závěr je demonstrováno použití frameworku v kontextu ukázkové aplikace, kde je pomocí Spring Data řešena realizace jednoduchých typových úloh.

Annotation

Title: Spring Data Framework as data access solution for Java Enterprise applications

The goal of this Diploma Thesis is to acquaint the reader with building of application data access layer using Spring Data framework. The initial part of thesis presents the theoretical background that will help better understand the issue being addressed. Subsequently is discussed the data access layer and are described it‘s basic implementation options referencing to known design patterns. The main part contains a description of Spring Data framework that focus on it's general modul reffered as Spring Data Commons. The last part contains a demonstration of framework usage in the context of sample application where the Spring Data is helping to realize simple use cases.

Obsah

1 Úvod

Tato diplomová práce se bude zabývat frameworkem Spring Data, který představuje ucelené řešení pro vytváření aplikační vrstvy pro přístup k datům v Java Enterprise aplikacích. Jedná se především o vyčlenění vrstvy pro přístup k datům, které přispívá v moderních Java EE aplikacích k přehlednější a snadnější údržbě programového kódu a celkově tak napomáhá k vyšší životnosti systémů. Navíc v kontextu rostoucího počtu databázových technologií označovaných jako NoSQL, které se výrazně liší od klasického pojetí, kterým je koncept relační databáze, dochází k výrazným změnám na úrovni architektury aplikace.

Objevují se dnes systémy, které k vykonávání své činnosti běžně používají několik perzistenčních technologií. V takovýchto podmínkách dostává vrstva pro přístup k datům ještě větší význam. V souvislosti s tím se také softwaroví architekti a vývojáři musí vypořádat s novými požadavky, které se na vrstvu pro přístup k datům objevují. Stále přetrvává potřeba oddělit prvky pro přístup k datovému zdroji od ostatních částí aplikace. Nově se ale objevuje snaha udržet rozhraní, která specifické databázové technologie zapouzdřují, co nejvíce jednotná. Umožnit tak vývojáři oprostit se od vědomosti, která konkrétní technologie na pozadí aplikace běží, a použít známé rozhraní. Dalším významným požadavkem na vrstvu pro přístup k datům, který se může jevit v konfliktu s představou jednotného rozhraní, je snaha maximálně využít potenciálu zvolené perzistenční technologie. Jedná se zde o specifické vlastnosti, které dělají zvolenou technologii výjimečnou, a proto není žádoucí tyto přednosti ignorovat a před vývojářem je skrývat. Framework Spring Data tento moderní přístup podporuje. Nabízí vývojáři známé rozhraní, kterým však neomezuje plnohodnotné použití zvolené perzistenční technologie.

Tato práce si dává za cíl představit, jakým způsobem vybudovat vrstvu pro přístup k datům pomocí frameworku Spring Data. Bude zde rozebrána obecná část frameworku, označovaná jako Spring Data Commons, která poskytuje základ pro technologicky specifické distribuce frameworku. Po prostudování této práce by si měl čtenář udělat konkrétní představu o implementaci vrstvy pro přístup k datům, a pokud má zkušenost s vývojem Java EE aplikací, měl by být schopen její implementace pomocí Spring Data frameworku.

2 Vrstva přístupu k datům

Na úvod této kapitoly budou představena teoretická východiska, která poslouží k lepšímu pochopení ústředního tématu, kterým je vrstva pro přístup k datům.

2.1 Data, databáze

Chris Date ve své publikaci [L-1] podrobně rozebírá definici základních pojmů, kterými jsou databázový systém a databáze. Dále zde prezentuje svůj pohled na data a informace. Ačkoliv si je vědom toho, že někteří autoři, jako například Vilém Sklenák v [L-2], používají pojem data pro označení toho, co je ve skutečnosti uloženo v databázi a pojem informace pro odkázání se na význam těchto dat, ve své publikaci mezi nimi nedělá rozdíl a chápe je jako synonyma.

Databázový systém je definován jako počítačový systém uchovávající záznamy. Cílem tohoto systému je uchovávat informace a umožnit uživatelům jejich získání a modifikaci na vyžádání. Informací může být cokoliv, co má význam pro uživatele, bez ohledu na to, jestli jím je organizace nebo jednotlivec. Databáze je pak souhrn perzistentních dat, která jsou používána aplikačními systémy určitého subjektu. Subjektem může být opět organizace nebo jedinec. Jako perzistentní data jsou označena data trvalá, která se svou povahou liší od dat dočasných. Dočasná data jsou na rozdíl od perzistentních přechodné povahy.

2.1.1 Relační databáze

První databázovou technologií, která zde bude představena, je klasická relační databáze. Jak píše Edgar Codd v [L-3], výchozím konceptem pro tento druh databáze je matematická relace, na jejímž základě je postaven relační model. Chris Date dále v [L-1] popisuje tři základní aspekty tohoto modelu.

Prvním je strukturální aspekt, který shrnuje, že data v databázi jsou uživatelem vnímána jako tabulky a nic jiného než tabulky. Řádek v tabulce odpovídá jednomu záznamu. Druhým v pořadí je integritní aspekt, který odkazuje na integritní omezení dat, jejichž účelem je zabránit výskytu nežádoucích a nereálných stavů. Například každý sloupec má přesně daný datový typ a nelze do něj vložit jinou než odpovídající hodnotu. Třetím a posledním je manipulativní aspekt. Ten poukazuje na základní operátory pro manipulaci s daty v tabulkách, které má uživatel k dispozici. Jedná se o

operátory umožňující odvozování nových tabulek z již existujících. Mezi tři nejdůležitější patří selekce pro výběr podmnožiny řádků z tabulky, projekce pro výběr podmnožiny sloupců z tabulky a spojení pro propojení dvou tabulek přes určitou vlastnost.

Jako jeden z hlavních faktorů úspěchu prosazení relačních databází do komerčního sektoru označuje Ramez Elmasri v [L-4] dotazovací jazyk Structured Query Language (dále jen SQL), který se stal v tomto ohledu standardem pro komunikaci klienta s databází. Tvrdí, že se jedná o jazyk deklarativní, což znamená, že uživatel jednotlivými příkazy specifikuje pouze to, co má být výsledkem dotazu, nechávaje optimalizaci a rozhodnutí o tom, jak dotaz provést, na databázovém systému. Dále popisuje, že SQL je komplexní databázový jazyk, v rámci kterého je možné identifikovat Data Definition Language (zkráceně DDL) a Data Manipulation Language (zkráceně DML). Příkazy DDL slouží k definici datových struktur. Příkazy DML jsou určeny k dotazování a provádění modifikací dat. Kromě DDL a DML vyčleňuje Ramez Elmasri další části jazyka SQL, kterými jsou příkazy pro definici pohledů, specifikaci zabezpečení, nastavení integritních omezení a provádění transakcí.

Posledním bodem, který zde bude v souvislosti s relačními databázemi zmíněn, jsou ACID vlastnosti transakcí. Zkratka ACID odkazuje na vlastnosti Atomic, Consistent, Isolation a Durable (v překladu: atomičnost, konzistence, izolovanost a trvalost).

Mark Grand v [L-5] tyto vlastnosti popisuje. Atomičnost znamená, že změny, které transakce provádí, jsou atomické. Jinými slovy, v rámci transakce buď dojde k provedení všech změn, a to právě jednou, nebo neproběhne změna žádná. Konzistence určuje, že transakce je korektní transformací stavů. To znamená, že stav na začátku a na konci transakce je vždy konzistentní dle integritních omezení, bez ohledu na její úspěch nebo selhání. Izolovanost definuje chování transakcí z pohledu konkurentního spuštění. I když dojde k současnému spouštění dvou transakcí, navzájem se jeví jako sériově spuštěné. Jedna transakce vnímá provedení té druhé před nebo po sobě, nikoliv současně. Trvalost vypovídá o tom, že jakmile je transakce úspěšně dokončena, provedené změny se stávají perzistentními.

2.1.2 NoSQL databáze

Druhou skupinou databázových technologií, která zde bude představena, je NoSQL. Jak už samotný název napovídá, definice této skupiny bude pramenit ve vymezení oproti klasickým SQL databázím. Shashank Tiwari ve své publikaci [L-6] tuto definici detailněji rozebírá. Tvrdí, že tvůrci pojmu NoSQL (v překladu: ne SQL) dali přednost marketingově lépe znějícímu slovu NoSQL, které však přesně nevystihuje tuto skupinu technologií. Z pohledu vlastností pak spíše připadají v úvahu výstižnější slova, jako například NoRDBSM nebo NoRelational, které odkazují přímo na relační databáze. Případně zde doporučuje pojem NoSQL chápat jako zkratku Not Only SQL (v překladu: ne pouze SQL). Bez ohledu na přesný význam slova NoSQL se dnes pod tímto pojmem rozumí databáze a přidružené technologie, které svým konceptem nenásledují tradiční zavedené principy relačních databázových systémů.

Pro relační databáze byly v předchozí kapitole popsány základní vlastnosti ACID. Pro NoSQL technologie v tomto ohledu existuje obdoba označovaná jako BASE. Martin Fowler a Sadalage Pramod v publikaci [L-7] tyto vlastnosti zmiňují. Zkratka BASE odkazuje na vlastnosti Basically Available, Soft state, Eventual consistency (v překladu: v podstatě dostupné, měkký stav, eventuálně konzistentní). Autoři publikace však tvrdí, že tyto vlastnosti nebyly řádně definovány, a proto jim vzhledem k jejich volnosti nepřiřazují velký význam, na rozdíl od ACID vlastností relačních databází.

Vzhledem k obecnému vymezení definice NoSQL systému existuje celá řada technologií, která do této skupiny spadá. Z tohoto důvodu zde bude představeno základní rozdělení podle použitého datového modelu, které představil Martin Fowler v [I-1].

První a zároveň nejjednodušší model, který bude představen, se nazývá key-value (klíč-hodnota). Jak už jeho název napovídá, struktura uložených dat se v tomto případě skládá ze dvou částí. První částí je známý klíč, který slouží k jednoznačné identifikaci. Druhou část pak tvoří neznámá hodnota, jejíž strukturu databázový model nijak nepopisuje. Může se jednat o jednoduché číslo, komplexní dokument nebo obrázek. Databáze chápe hodnotu jako nedělitelný celek a její zpracování přenechává na klientské aplikaci. Databáze je pak v tomto ohledu schopna realizovat

pouze elementární operace, typicky vrácení hodnoty pro daný klíč. Některé technologie založené na tomto modelu mohou obsahovat metadata, prostřednictvím kterých se lze odkazovat na části struktury v rámci hodnoty, čímž se od tohoto konceptu odchylují. Příkladem použití key-value modelu je technologie Redis [I-2].

Druhým v pořadí je takzvaný document (dokument) model. Obdobně jako v key-value modelu, ani zde neexistuje schéma popisující strukturu dat. Navzdory absenci schématu jsou však perzistovaná data uchovávána ve strukturované formě v podobě dokumentu. To v sobě zahrnuje možnost dotazování se pouze na relevantní části dat. Samotné dotazování je pak založeno na předpokladu. Bez schematického popisu totiž neexistuje jistota, že dotazovaný dokument obsahuje požadovaný atribut. S tímto omezením, které je také označováno jako implicitní schéma, se musí programátoři, používající dokumentově orientované databáze, vyrovnat. Příkladem tohoto přístupu je technologie MongoDB [I-3].

Třetím představeným je column-family (sloupcový) model. Identifikátor v tomto případě odkazuje na skupinu sloupců, která tvoří agregovaný výsledek. Každý sloupec je dále rozdělen na název a hodnotu. Jednotlivé sloupce mohou být pro různé řádky odlišné. Oproti ostatním představeným je tento sloupcový model komplexnější a umožňuje snadno získávat pouze relevantní část dat. Příkladem technologie založené na tomto modelu je Apache HBase [I-4].

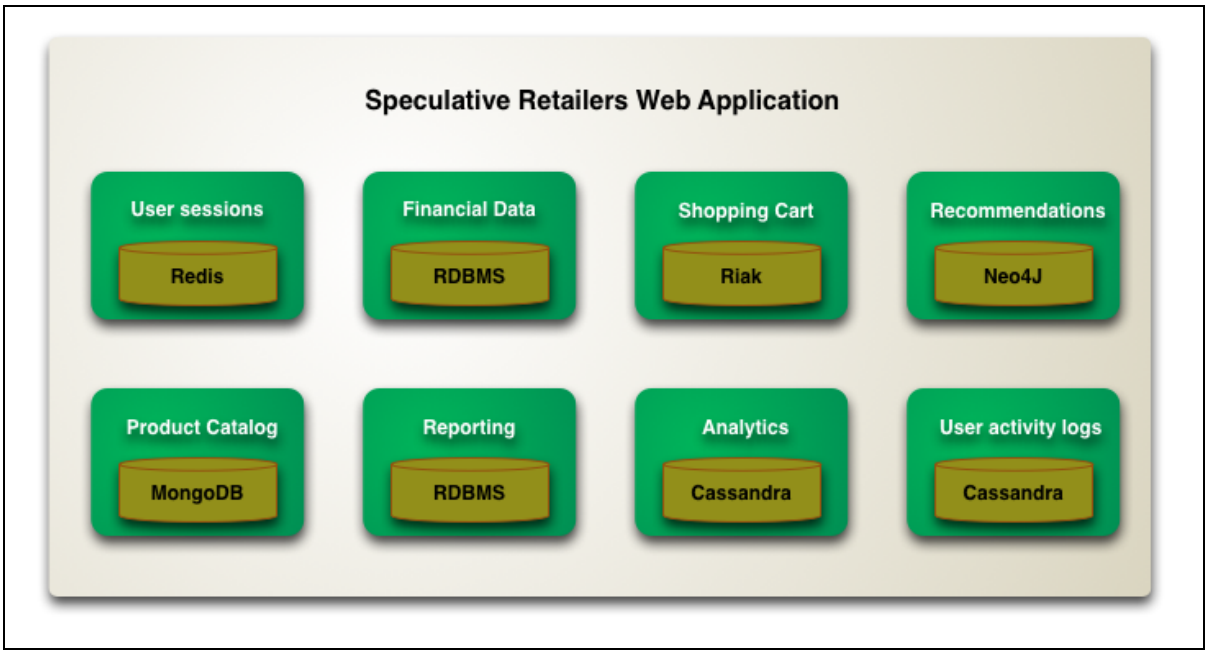
Čtvrtým, a zároveň posledním představeným, je graph (grafový) model. Struktura modelu je tvořena uzly a hranami. Na rozdíl od všech předchozích zmíněných konceptů grafový model neagreguje data. Zaměřuje se spíše na menší počet záznamů s velkým množstvím vzájemných propojení. Díky svým vlastnostem umožňují grafové databáze vytvořit ideální prostředí k zachycení složitých vztahů. Jedná se zároveň o model, který splňuje ACID podmínky transakčních operací tak, jak je to popsáno u relačních databází. Představitelem těchto technologií je například Neo4j [I-5].

O konkrétním využití NoSQL databází podle typu modelu píše Mark Pollack v [L-8]. Tvrdí, že key-value databáze by měly být zvoleny pro realizaci funkcionalit, jako je vyrovnávací paměť – cache, dokumentové databáze pro perzistenci stromově reprezentovaných a agregovatelných datových struktur a grafové databáze pro vysoce navzájem propojená data.

2.1.3 Polyglot perzistence

V souvislosti s velkým počtem databázových technologií, které jsou dnes k dispozici, zde bude představen koncept systému, který se odchyluje od tradičního modelu, čímž je použití jednoho typu databáze. Takový systém používá několik různých databází pro uspokojení specifických potřeb od rychlosti vrácení požadovaných dat po potřebu jejich zachycení do složitých struktur a vztahů. Martin Fowler v [I-6] tento přístup, který označuje jako polyglot perzistenci, dále rozebírá. Tvrdí, že volba perzistenční technologie by měla především reflektovat způsob, jakým bude systém s daty manipulovat. A ačkoliv se mohou náklady spojené s učením se nových technologií a jejich provozem jevit jako vysoké, přidaná hodnota v podobě předcházení výskytu obecných problémů prostřednictvím volby vhodné technologie bude daleko větším přínosem. Datová základna totiž velmi často bývá úzkým hrdlem celého systému.

Martin Fowler a Sadalage Pramod v publikaci [I-7] představují návrh prodejního systému používající polyglot perzistenci. Model samotný obsahuje reference na konkrétní, dnes běžně dostupné databázové technologie. Ukázka tohoto systému je zachycena na obrázku 1.



Obrázek 1: příklad polyglot systému [I-7]

Vysvětlují, že pro ukládání uživatelských relací je vhodné použít key-value technologii Redis, s ohledem na rychlost výměny dat bez nutnosti trvalého uložení. Pro ukládání finančních dat volí relační databázi kvůli jejím transakčním vlastnostem a vhodné tabulkové struktuře. Pro reprezentaci nákupního košíku je v modelu použita key-value NoSQL databáze Riak [I-8], která garantuje vysokou dostupnost napříč několika vzdálenými lokacemi a umožňuje slučovat nekonzistentní zápisy dat, se kterými se v tomto případě počítá. Výběr doporučení a nabídek je realizován v grafové databázi Neo4J s ohledem na vysokou míru provázanosti zákazníků, produktů a objednávek. Dokumentová databáze MongoDB je doporučena k ukládání produktového katalogu, kde se očekává velké množství načítacích a malé množství zapisovacích operací. Na tvorbu reportů je navrženo použití relační databáze, protože ji lze snadno integrovat s existujícími nástroji pro reportování. Z výkonnostního hlediska je zvolena technologie Cassandra [I-9] pro provádění náročnějších analytických činností a stejně je tomu tak i v případě uchovávání logů uživatelské aktivity.

2.2 Data Access Layer

V této části práce bude představena vrstva pro přístup k datům Data Access Layer (zkráceně DAL). Jedná se o aplikační vrstvu, jejíž definici rozebírá Vishal Layka v publikaci [L-9]. Tvrdí, že podstatou této vrstvy je abstrakce použitého perzistenčního mechanizmu. Ten by měl být oddělen od ostatních komponent aplikace, které utvářejí servisní vrstvu. Zároveň je zde zdůrazněno, že rozhraní poskytované vrstvou pro přístup k datům by mělo být co nejjednodušší a na rozdíl od servisní vrstvy by jeho implementace neměla obsahovat žádné fragmenty byznysové logiky. Tok volání pro takto nastavenou architekturu systému je určen směrem od servisní vrstvy k vrstvě přístupu k datům. Jinými slovy, servisní vrstva může použít k realizování své byznys logiky vrstvu pro přístup k datům, ale obráceně to není možné.

Obdobný pohled na tuto aplikační vrstvu představují také Clarence Ho a Rob Harrop v publikaci [L-10]. Tvrdí, že vyčleněním vrstvy pro přístup k datům je poskytnuto ostatním komponentám aplikace standardní rozhraní pro ukládání a načítání dat. Dále zdůrazňují, že vynechání této vrstvy povede k rozprostření zdrojového kódu pro zpřístupnění dat napříč aplikací, což má často za následek duplikování kódu, který je nákladné udržovat. V dlouhodobém horizontu pak tento špatně spravovatelný kód vede nevyhnutelně k chybám aplikace.

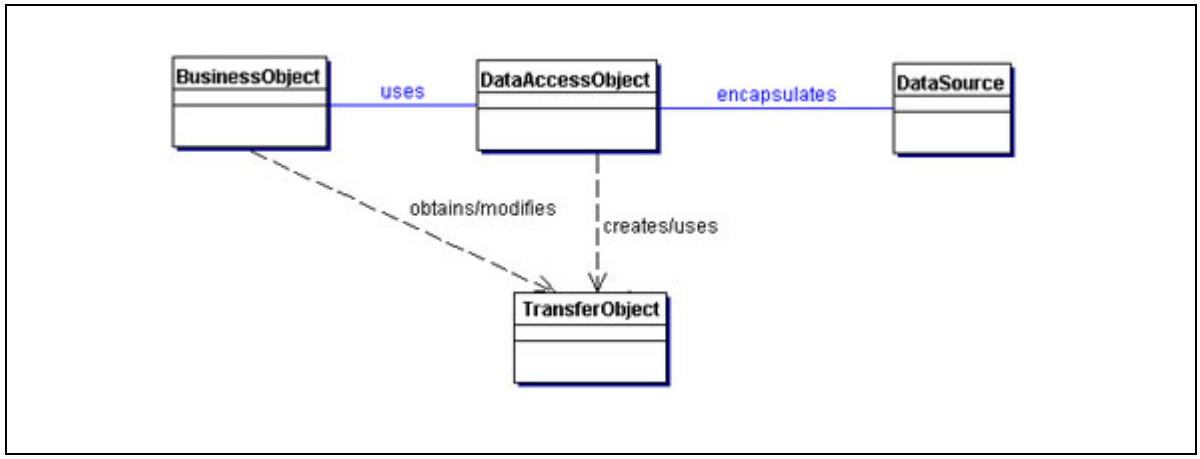
2.2.1 Návrhové vzory

V této kapitole budou ukázány vybrané návrhové vzory, pomocí kterých lze v Java Enterprise aplikacích vybudovat vrstvu pro přístup k datům. Jak uvádí [L-11], návrhový vzor systematicky pojmenovává, motivuje a vysvětluje obecný návrh, který řeší opakující se problémy v objektově orientovaných systémech. Vzor samotný popisuje problém, jeho řešení, kdy toto řešení použít a důsledky jeho použití. Také může obsahovat implementační doporučení a ukázky.

2.2.1.1 Data Access Object

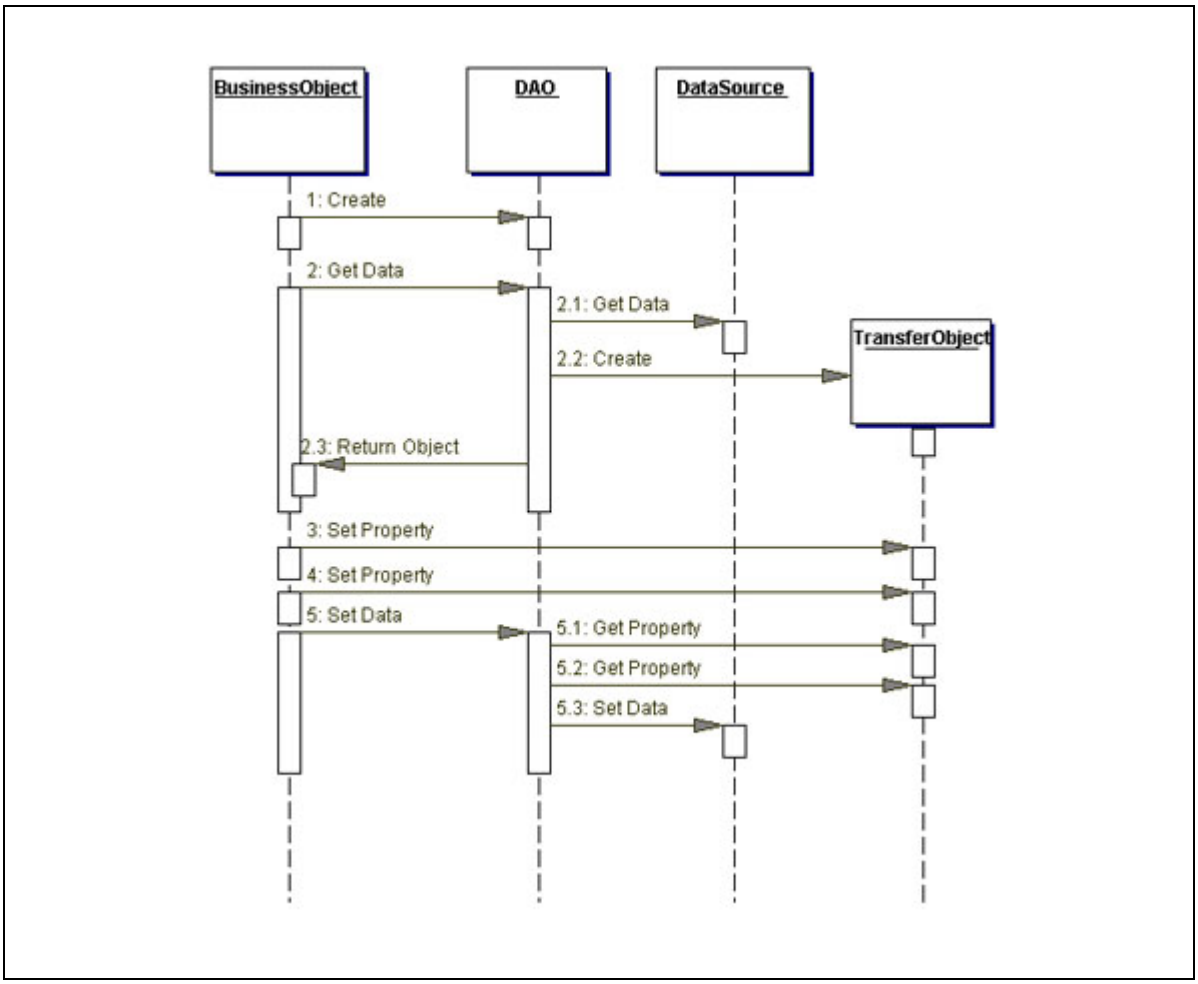
Jako první zde bude představen návrhový vzor označovaný jako Data Access Object (data zpřístupňující objekt, zkráceně DAO). Tento vzor je popsán v oficiální dokumentaci k Java Enterprise Edition [I-10]. Jeho princip spočívá v použití DAO k zapouzdření a abstrakci datových zdrojů. DAO je zodpovědný za obsluhu datového

zdroje, ukládání a získávání dat. Je na něj napojena klientská část aplikace, také označovaná jako Business Object, která využívá jeho rozhraní k odstínění technologicky závislé implementace. Klientská část nevidí do vnitřního mechanizmu DAO, kterým může být komunikace s relační databází nebo NoSQL technologií. Z důvodu tohoto zapouzdření není klient náchylný na úpravy v případě, že dojde ke změně implementace datového zdroje. Tyto změny se tak mohou adaptovat pouze na DAO, který v tomto ohledu figuruje jako adaptér mezi technologií datového zdroje a ostatními komponentami aplikace. Ukázka diagramu tříd, znázorňující základní komponenty, je představena na obrázku 2.



Obrázek 2: diagram tříd DAO [I-10]

Kromě již zmíněných aplikačních částí, kterými jsou DAO, Business Object a DataSource, je na diagramu znázorněn také Transfer Object. Jeho účelem je zapouzdření DAO výstupů načítaných z datového zdroje. Transfer Object v tomto kontextu je možné v některých publikacích nalézt také pod pojmem Data Transfer Object (zkráceně DTO). Znázornění chování DAO vzoru je zachyceno v sekvenčním diagramu na obrázku 3.

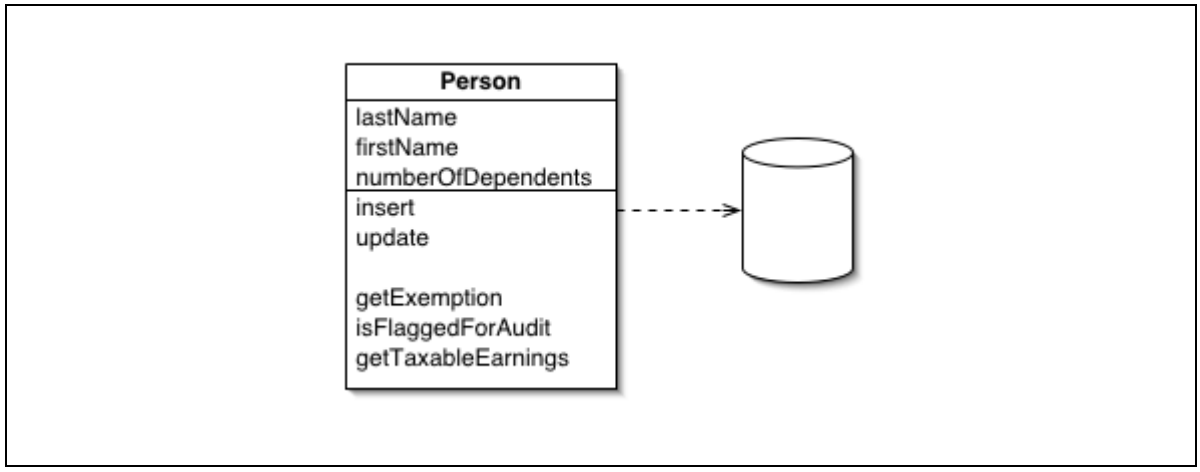


Obrázek 3: sekvenční diagram DAO [I-10]

Diagram rozebírá běžný aplikační scénář, kterým je načtení dat z datového zdroje, provedení lokální modifikace a jejich zpětné uložení. Krok 1 představuje získání instance DAO v klientské části – Business Object. Krok 2 reprezentuje zaslání požadavku na získání dat z Business Object na DAO. Krokem 2.1 je vlastní získání dat z datového zdroje. Jedná se o komunikaci v nativním jazyce datové technologie, kterou iniciuje DAO. Krok 2.2 představuje zapouzdření získaného výsledku do nově vytvořené instance DTO. Za vytvoření DTO je v tomto případě zodpovědný DAO. Krok 2.3 znamená dokončení načítání dat z pohledu klientské části, která nyní obdržela zapouzdřený výsledek z DAO. V kroku 3 a 4 dochází k manipulaci s DTO objektem, který je následně pozměněn. V rámci kroku 5 zasílá Business Object požadavek na DAO, aby došlo k uložení lokálně provedených změn DTO. V kroku 5.1, 5.2 DAO zjišťuje provedené změny. Krok 5.3 představuje provedení změn do datového úložiště.

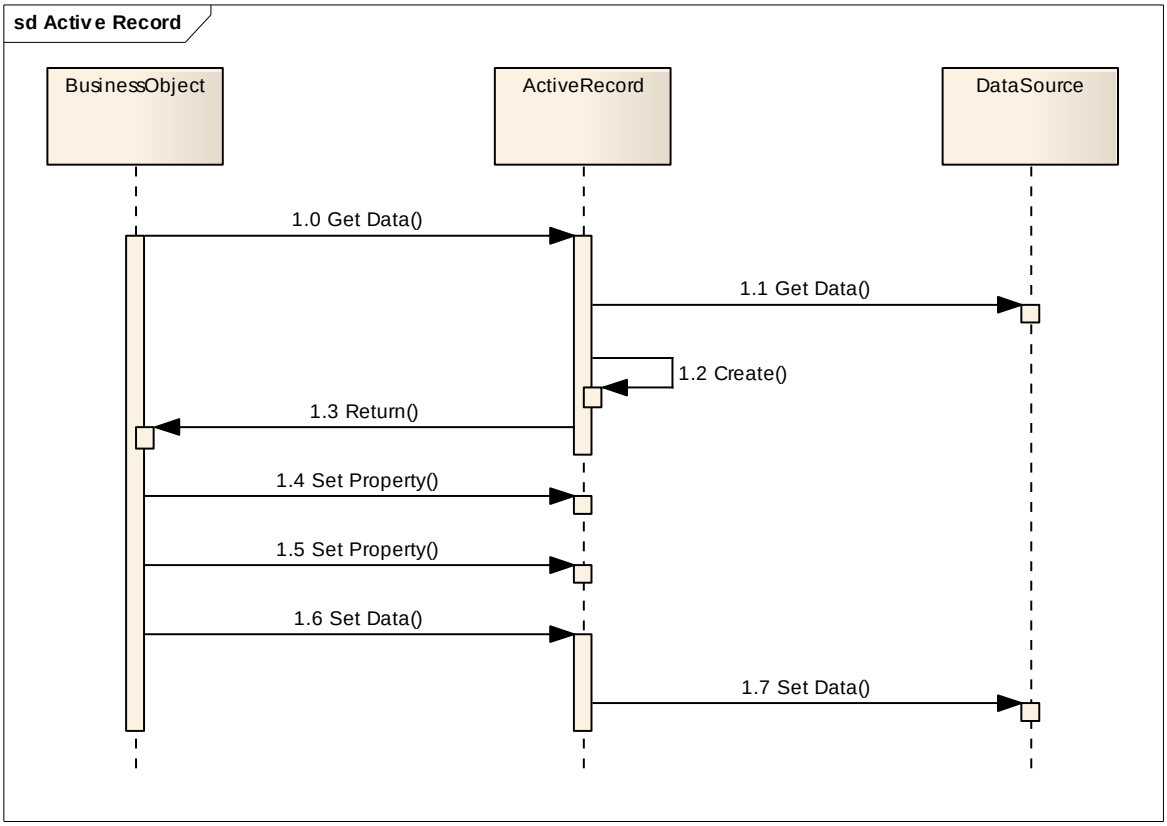
2.2.1.2 Active Record

Druhým návrhovým vzorem, který bude v souvislosti s vrstvou pro přístup k datům ukázán, je Active Record (aktivní záznam, zkráceně AR). Ve své publikaci [L-12] ho představuje Martin Fowler jako jeden ze vzorů architektury datových zdrojů aplikací. Vysvětluje, že každý objekt je definován prostřednictvím svého stavu a chování. V rámci aplikace je stav objektu vnímán jako data, která jsou perzistentní, a je potřeba zajistit jejich uložení do datového úložiště. Active Record v tomto ohledu používá nejvíce zřejmý přístup, kterým je přidání logiky pro přístup k datům přímo do doménového modelu. Doménový objekt pak zaobaluje jeden záznam v datovém úložišti a zapouzdřuje, kromě své vlastní doménové logiky, také přístup k datům. Důležitým předpokladem pro efektivní použití tohoto vzoru je vysoká míra shody struktury dat v datovém úložišti a doménového modelu. Ukázka návrhu doménové třídy podle vzoru Active Record je přiložena na obrázku 4.



Obrázek 4: diagram tříd Active Record [I-11]

Na diagramu je navržena třída Person, která kromě datových atributů a doménové logiky obsahuje i metody insert a update. Ty slouží k vložení nového a úpravě existujícího záznamu do přidruženého datového úložiště. Znázornění chování Active Record vzoru je zachyceno v sekvenčním diagramu na obrázku 5.



Obrázek 5: sekvenční diagram Active Record [autor]

Podobně jako tomu bylo v případě DAO, diagram rozebírá základní scénář, kterým je načtení dat z datového zdroje, provedení lokální modifikace a jejich zpětné uložení. Krok 1 představuje zaslání požadavku na získání dat, které iniciuje Business Object. V tomto případě se jedná o statické volání, které se neváže na instanci Active Record, ale přímo na třídu. Krokem 1.1 je vlastní získání dat z datového zdroje. Opět se jedná o statické volání na úrovni třídy, komunikace probíhá v nativním jazyce datové technologie. Krok 1.2 představuje zapouzdření získaného výsledku do nově vytvořené instance doménové třídy, která vystupuje jako Active Record. Krok 1.3 znamená dokončení načítání dat z pohledu klientské části. V kroku 1.4 a 1.5 dochází k manipulaci s doménovou částí Active Record objektu, který je následně pozměněn. V rámci kroku 1.6 zasílá Business Object požadavek přímo na Active Record objekt k uložení lokálně provedených změn. Krok 1.7 představuje provedení změn do datového úložiště.

Ačkoliv Fowler popisuje Active Record výhradně v souvislosti s použitím relační databáze jako datového úložiště, tento vzor je možné aplikovat také na NoSQL technologie. Příkladem je projekt Mongoid [I-12], jehož filosofie je definována následovně:

„Filosofií Mongoid je poskytnout známé aplikační rozhraní pro vývojáře programující v jazyce Ruby, kteří používají vzory Active Record nebo Data Mapper, v kontextu využití potenciálu bez schematické, výkonné, dokumentově orientované, dynamicky dotazovatelné a atomické operace pro úpravy umožňující technologie MongoDB.“

3 Spring Data

V předchozí kapitole bylo popsáno, proč je vhodné v aplikaci vyčlenit vrstvu pro přístup k datům, a jaké jsou základní možnosti její realizace. Nyní bude detailněji představen jeden z předních frameworků, který se touto oblastí zabývá – Spring Data. Ještě před samotným popisem je však vhodné se seznámit se Spring Framework.

3.1 Základní popis Spring Framework

Cílem této kapitoly je objasnit původ a zařazení frameworku Spring Data. Ten totiž, jak už samotný název napovídá, spadá do skupiny technologií s označením Spring. Jak píše Rod Johnson ve svém článku s názvem Introduction to the Spring Framework [I-13], vývojáři Spring se již od roku 2003 zabývají vývojem technologií usnadňující tvorbu profesionálních aplikací a systémů založených na platformě Java.

Pro řešení od Spring bývá charakteristická především nízká provázanost jednotlivých částí, a tím i celkově vysoká modularita. Díky těmto vlastnostem je zde možnost použít pouze vybrané moduly Springu a zkombinovat je s technologiemi třetích stran tak, aby bylo dosaženo co nejvyšší přidané hodnoty. Ani modul Spring Data není v tomto konceptu výjimkou.

V současné době jsou pod označením Spring dostupná řešení viz tabulka 1 převzato z [I-14].

Tabulka 1: Spring projekty

Název	**Popis**
Spring Boot	Umožňuje snadnou tvorbu aplikací s orientací na minimum konfigurace a snadné spouštění.
Spring Framework	Představuje úplný základ Springu – obsahuje podporu pro návrhový vzor Dependency Injection, transakční zpracování, webové aplikace, atd.
Spring XD	Zjednodušuje vývoj aplikací pro práci s velkými objemy dat pomocí analytických dávkových úloh, adresování a exportu dat.
Spring Data	Poskytuje jednotný koncept pro přístup k datům v rámci relačních, ne-relačních i dalších technologií.
Spring Integration	Implementace známého konceptu Enterprise Integration Patterns, obsahuje zjednodušení zasílání zpráv a deklarativní adaptéry.

Spring Batch	Zjednodušuje a optimalizuje činnost zpracování velkoobjemových dávkových operací.
Spring Security	Implementace zabezpečení aplikace s komplexní a rozšiřitelnou autentizací a autorizací.
Spring HATEOAS	Zjednodušuje vytváření REST aplikací, které jsou založeny na HATEOAS principu.
Spring Social	Snadné propojení aplikace se softwarem třetích stran – jako je Facebook, Twitter, LinkedIn a další.
Spring AMQP	Aplikuje základní koncepty Springu na vývoj aplikací založených na AMQP zasílání zpráv.
Spring Mobile	Zjednodušuje vývoj webových aplikací pro mobilní zařízení prostřednictvím detekce zařízení a progresivních renderovacích možností.
Spring for Android	Poskytuje klíčové komponenty Springu pro vývoj na platformě Android.
Spring Web Flow	Podpora pro vývoj webových aplikací s kontrolovanou navigací jako je například odbavení letu nebo vyřizování žádosti o úvěr.
Spring Web Services	Usnadňuje vývoj webových služeb s definovaným rozhraním jaké má například protokol SOAP.
Spring LDAP	Usnadňuje vývoj aplikací používajících systém LDAP pomocí pro Spring typického přístupu – šablonování.
Grails	Postavená na Springu, tato technologie poskytuje plnohodnotné prostředí pro vytváření webových aplikací v jazyce Groovy.
Groovy	Groovy přináší vysokou efektivitu do JVM, která v sobě zahrnuje podporu pro statické a dynamické programování, skriptování a doménové jazyky.
Spring Scala	Spojuje výrazové možnosti jazyka Scala dohromady s výhodami a rozsahem Springu.
Spring Roo	Umožňuje rychle a jednoduše vygenerovat plnohodnotné Java aplikace v rámci několika minut.
Spring BlazeDS Integration	Poskytuje prvotřídní podporu pro používání Adobe BlazeDS ve Spring aplikacích s Adobe Flex klienty.
Spring Loaded	Zvyšuje produktivitu vývoje pomocí zpropagování změn ve zdrojovém kódu prostřednictvím JVM aplikace v čase, kdy změny nastanou.

Spring Shell	Poskytuje solidní základ pro vytváření konzolových aplikací založených na programovacím modelu Springu.
Rest Shell	Ulehčuje psaní a testování REST aplikací s vyhledáváním zdrojů a interakcí pomocí příkazové řádky.

Z pohledu použitelnosti je výhodné, že Spring a všechny jeho součásti spadají pod open source licenci Apache Licence 2.0 [I-15]. Ta totiž umožňuje právně bezproblémové použití frameworku jak v soukromých, tak v komerčních aplikacích. Jedná se tedy o svobodný software, který je díky tomu používán mnoha vývojáři, a tím buduje vlastní komunitu, jejíž členové doplňují oficiálně poskytované materiály o další specifikace a návody. V konečném důsledku je tvorba a ladění chyb na Spring projektech pro vývojáře ještě komfortnější.

Pro tvorbu Spring aplikací je možné použít vývojové prostředí Spring Tool Suite (které je také známé pod zkratkou STS). Základem STS je vývojové prostředí Eclipse IDE [I-16], doplněné o řadu pluginů pro práci se Spring projekty, včetně Spring Data.

3.2 Historie a koncept Spring Data

Následující text bude zaměřen na historii a okolnosti vzniku frameworku Spring Data. Ústředním bodem pak bude koncept, také by se dalo říct myšlenka nebo důvod vzniku. Podkladem pro tuto kapitolu je kniha od Marka Pollacka [L-8].

Počátky tohoto projektu sahají do roku 2010, kdy se Rod Johnson ze Spring a Emil Eifrem z Neo Technologies pokoušeli integrovat do Springu napojení na grafovou databázi Neo4j. Během jejich implementace se jim podařilo aplikovat nový, zcela odlišný přístup k této problematice. Výstupem pak byl zdrojový kód, který lze považovat za jednu z prvních verzí Neo4j modulu pro Spring Data.

Odlišnost jejich nového přístupu je jasně viditelná v kontrastu s tradiční podobou podpory přístupu k datům ve Spring. Je nutné si uvědomit, že tou dobou již Spring obsahoval sofistikovanou podporu pro běžnou práci s tradičními technologiemi pro přístup k datům. Ať už se jedná o prosté JDBC, nativní ORM frameworky jako například Hibernate nebo myBatis, nebo přímo standard JPA. Vlastní podpora se v tomto případě převážně zaměřuje na usnadnění konfigurace celé infrastruktury, správu zdrojů a závislostí a v neposlední řadě také na zachytávání a překlad chyb do lidsky lépe čitelné formy. Tato podpora se v průběhu let ustálila a zásahy prováděné do této vrstvy jsou v současné době minimální. Podpora tradičních, relačních technologií pro přístup k datům se pro Spring stala v této oblasti prioritou, neboť se jednalo o převládající technologii realizující persistenci dat.

Neo4j je však grafová databáze, spadající do skupiny s označením NoSQL. Tato skupina se navíc v poslední době těší stále většímu zájmu. Představuje totiž vhodnou alternativu persistence dat tam, kde by použití relační databáze vedlo k dosažení jejího limitu nebo omezení. S rostoucím zájmem o NoSQL technologie však bylo zapotřebí vyřešit jednu klíčovou otázku – jak a kam integrovat podporu NoSQL databází pro Spring vývojáře.

Ještě před odpovědí na tuto otázku je nutné si uvědomit, že zde není pouze významný rozdíl mezi SQL a NoSQL databázemi. Významné rozdíly najdeme i mezi jednotlivými technologiemi v rámci NoSQL. Zatímco jednotlivé relační databáze se v globálním pohledu liší pouze svými dialekty, v případě NoSQL jsou zde zásadní odlišnosti už na úrovni základních vlastností a možností konkrétní databáze. To se

následně zohledňuje i ve výsledné infrastruktuře nabízeného API. Dalo by se tedy říct, že co NoSQL databáze - to vlastní API.

Ale zpět k původní otázce. Nyní by již mělo být jasné, že volba jednotného API pro konfiguraci a infrastrukturu není tou správnou cestou. Ukazuje se, že výsledný způsob integrace bude muset umožnit pro danou technologii specifickou konfiguraci a specifickou formu použití jejích možností. Nedodržení tohoto předpokladu a snaha unifikovat práci s NoSQL databázemi přes jednotné API by v koncovém důsledku znamenala dobrovolně se vzdát výhod dané technologie.

Z výše uvedeného vyplývá, jak by měla a jak by neměla vypadat integrace NoSQL databází. Nyní zbývá ještě vyřešit otázku, kam tuto integraci zahrnout. Pravděpodobně první místo, kam by Spring vývojář tuto integraci implementoval je JPA, které tvoří vývojový standard pro tvorbu perzistenční vrstvy aplikací v Javě. V tomto případě však lze narazit na několik překážek. První překážka vyplývá už ze samotné specifikace standardu – JSR-317 [I-17], kde se v úvodní části píše:

„Tento dokument představuje specifikaci Java API pro správu perzistence a objektově-relačního mapování v Java EE a Java SE. Technickým cílem této práce je poskytnout vývojářům Java aplikací objektově-relační mapovací nástroj založený na použití Java doménového modelu k managementu relační databáze.“

V předchozím odstavci je jasně definováno, že JPA je orientováno výhradně na objektově-relační mapování – tedy použití relačních SQL databází. Pokud budou tato úvodní slova ignorována, a místem pro integraci NoSQL bude vybráno JPA, je tu další překážka. Koncept relačních databází se totiž v JPA specifikaci odráží od začátku dokumentace až do jejího konce. Definují se zde pojmy, které jsou hluboce spjaty se světem relačních databází – např. anotace @Table, @Column, které pro NoSQL databáze nedávají vůbec smysl, nebo také transakční zpracování, které pro NoSQL není podmínkou. Díky této vysoké míře nekompatibility lze celkově říct, že specifikace JPA nedává volný prostor pro podporu NoSQL databází a není proto vhodným místem pro jejich integraci.

Oproti tomu zde existuje prostor pro nový projekt – Spring Data, který by se po vzoru integrace Neo4J dal použít i pro ostatní NoSQL databáze. I z pohledu definice cílů [I-18] Spring Data je to ideální místo pro tento záměr:

„Spring Data poskytuje známý a konzistentní programovací model používaný ve Spring, který integruje použití NoSQL a relačních technologií, aniž by došlo ke ztrátě specifických vlastností a výhod dané technologie.“

Tato definice zároveň rozšiřuje možnosti uplatnění Spring Data nejen na NoSQL ale i na relační databáze. Tyto dvě oblasti dohromady tvoří hlavní motivaci pro další vývoj projektu Spring Data, který se skládá ze specializovaných modulů jak pro NoSQL, tak i pro SQL a relační databáze. Více informací o konkrétních podporovaných technologiích řeší následující kapitola.

3.3 Distribuce Spring Data

Ještě než bude Spring Data detailněji představen, je vhodné se podívat na způsob, jakým je tento framework distribuován.

Samotné Spring Data se člení do několika projektů podle technologie, nad kterou se staví. Každý Spring Data projekt se zabývá vždy konkrétní technologií, jak zobrazují přiložené tabulky převzaté z oficiálních zdrojů [I-19]. Tabulka 2 odkazuje na hlavní projekty Spring Data.

Tabulka 2: hlavní projekty Spring Data

Hlavní projekty	
Spring Data JPA	Podpora pro objektově-relační přístup prostřednictvím Java standardu JPA.
Spring Data MongoDB	Podpora objektově-dokumentového přístupu pro MongoDB.
Spring Data Neo4j	Podpora objektově-grafového přístupu v rámci použití Neo4j.
Spring Data Redis	Podpora úložiště založeného na přístupu klíč-hodnota v rámci technologie Redis.
Spring for Hadoop	Podpora distribuovaného souborového systému HDFS, přístupu MapReduce a rozšíření Pig a Hive [I-20].
Spring Data GemFire	Podpora platformy pro distribuovanou správu dat v GemFire.
Spring Data REST	Umožňuje vystavení Spring Data rozhraní přes REST HTTP.
Spring Data JDBC extension	Rozšiřuje možnosti použití JDBC o QueryDSL a pokročilé programování v Oracle.

Tabulka 3 odkazuje na komunitní projekty. Jedná se o projekty, na jejichž vývoji se aktivně podílí komunita Springu.

Tabulka 3: komunitní projekty Spring Data

Komunitní projekty	
Spring Data Solr	Standardní podpora pro Apache Solr.
Spring Data Couchbase	Standardní podpora pro Couchbase.
Spring Data Elasticsearch	Standardní podpora pro Elasticsearch.
Spring Data Cassandra	Standardní podpora pro technologii Cassandra
Spring Data DynamoDB	Standardní podpora pro DynamoDB

Zavedení Spring Data do projektu je možné několika způsoby. Od ručního přiložení JAR souborů přímo do cílové aplikace, až po použití systémů pro správu závislostí jako je například Apache Maven nebo Gradle. Z pohledu profesionálního vývoje aplikací pak připadá v úvahu spíše druhá varianta, proto je zde přiložen příklad konfigurace pro Spring Data JPA verze 1.4.3.RELEASE. Kód 1 ukazuje přidání této závislosti v Apache Maven.

```
<dependencies>
	<dependency>
		<groupId>org.springframework.data</groupId>
		<artifactId>spring-data-jpa</artifactId>
		<version>1.4.3.RELEASE</version>
	</dependency>
</dependencies>
```

Kód 1: závislost Spring Data JPA – Maven

Kód 2 znázorňuje obdobu pro nástroj Gradle.

```
dependencies {

	compile 'org.springframework.data:spring-data-jpa:1.4.3.RELEASE'

}
```

Kód 2: závislost Spring Data JPA – Gradle

3.4 Spring Data Commons

Ústředním bodem kapitoly je modul Spring Data Commons, který představuje koncepční základ napříč Spring Data projekty. Modul je také samostatně zdokumentován v [I-21] a verze 1.6.4-RELEASE je hlavním podkladem této kapitoly. Aby bylo dosaženo celistvého výkladu, obsahuje popis Spring Data Commons příklady a drobné přesahy, které se váží na konkrétní distribuci – Spring Data JPA.

3.4.1 Repository rozhraní

První velmi důležité rozhraní, které zde bude představeno, se nazývá Repository. Jedná se o takzvané značkovací rozhraní. To znamená, že rozhraní nedefinuje žádné metody, které by bylo potřeba v případě implementace doplnit. Na druhou stranu se jedná o rozhraní generické, které od své implementace vyžaduje specifikaci dvou generických typů. Prvním typem v pořadí, označeným písmenem T, je specifikace doménového objektu, pro který bude implementace Repository realizovat obsluhu. Druhým typem, označeným ID, je specifikace identifikace. Datový typ identifikace musí navíc splňovat ještě jednu podmínku – sám musí implementovat značkovací rozhraní Serializable. Účel Repository rozhraní je především v zachycení těchto dvou typů, se kterými se dále pracuje, a také v usnadnění nalezení dalších rozšiřujících rozhraní.

Další rozhraní, které zde bude popsáno, se nazývá CrudRepository. To rozšiřuje jednoduché rozhraní Repository o základní metody pro realizaci operací označovaných zkratkou CRUD. Jedná se o CREATE, READ, UPDATE, DELETE. Operace označené jako CREATE slouží k ukládání nově vytvořených záznamů. Operace READ slouží ke čtení exitujících záznamů. Operace UPDATE realizuje modifikace/úpravy existujících záznamů. Operace DELETE slouží k odstranění existujících záznamů. Provázanost těchto operací a metod z CrudRepository je znázorněna v přiložené tabulce 4.

Tabulka 4: Provázanost CRUD operací a metod CrudRepository

Typ operace	**Metody z rozhraní CrudRepository<T, ID>**
CREATE	<S extends T> S save(S entity) <S extends T> Iterable<S> save(Iterable<S> entities)
READ	T findOne(ID id) boolean exists(ID id) Iterable<T> findAll() Iterable<T> findAll(Iterable<ID> ids) long count()
UPDATE	<S extends T> S save(S entity) <S extends T> Iterable<S> save(Iterable<S> entities)
DELETE	void delete(ID id) void delete(T entity) void delete(Iterable<? extends T> entities) void deleteAll()

Zajímavostí je překrytí operací CREATE a UPDATE, jak ukazuje přiložená tabulka. Obě tyto operace totiž na straně CrudRepository realizují stejné metody s označením save.

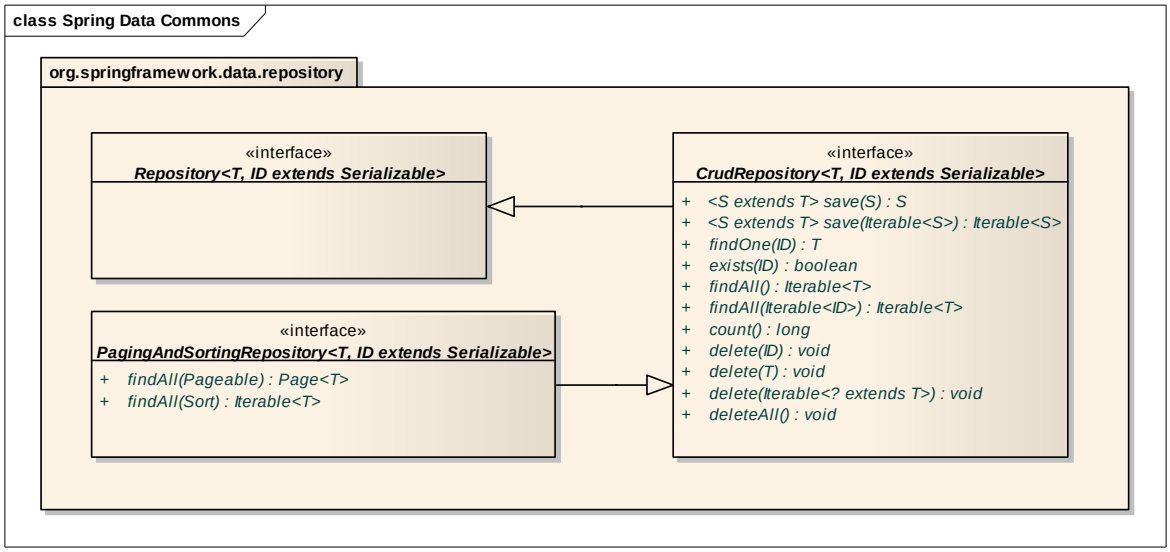
Poslední ze základních rozhraní je PagingAndSortingRepository, které rozšiřuje CrudRepository a přidává k němu metody pro práci se stránkováním a řazením. Důvodem potřeby tohoto rozšíření je velké množství neutříděných dat na jedné straně a omezení CrudRepository, které pro READ operace na úrovni získání vícero záznamů nabízí pouze metodu findAll, na straně druhé. Metoda findAll totiž iteruje pouze po jednom záznamu a neumožňuje jiné řazení než podle ID. Z toho důvodu byly přidány dvě rozšiřující metody, které tento problém řeší. Jejich provázanost na operace PAGING a SORTING (stránkování a řazení) je znázorněna v tabulce 5.

Tabulka 5: provázanost operací PAGING a SORTING a metod PagingAndSortingRepository

Typ operace	**Metody z rozhraní PagingAndSortingRepository<T, ID>**
PAGING	Page<T> findAll(Pageable pageable)
SORTING	Iterable<T> findAll(Sort sort)

Použití nových metod PagingAndSortingRepository dále odkazuje na dvě nová rozhraní – Pageable, Page a třídu – Sort. Pageable je využíváno při zadávání vstupů. Konkretizuje se tak počet záznamů na jednu stránku a pořadí stránky, která má být získána. Rozhraní Page pak slouží k zapouzdření výsledku takto provolaných metod. Pomocí třídy Sort lze předefinovat implicitní způsob řazení, který je nastaven podle ID. Je zde možnost definovat pořadí atributů, podle kterých chceme řadit, a také směr řazení – vzestupně nebo sestupně.

Celkový pohled na hierarchii základních rozhraní je znázorněn v diagramu tříd na obrázku 6.



Obrázek 6: základních rozhraní Spring Data Commons [autor]

Kromě těchto základních rozhraní, která jsou k dispozici bez ohledu na použitou perzistenční technologii, obsahují specifické distribuce Spring Data obvykle i svá vlastní Repository rozhraní. Ta pak nabízejí rozšiřující možnosti použití, které umožňují plnohodnotné využití výhod dané technologie. Existují tak rozhraní jako například JpaRepository pro JPA, MongoRepository pro MongoDB nebo GraphRepository pro Neo4J a další.

3.4.2 Konfigurace

Konfigurace obvykle představuje klíčovou roli pro správné fungování frameworku. Jinak tomu není ani v případě Spring Data, ačkoliv samotná konfigurace je zde velmi minimalistická. Jedinou povinnou částí je totiž definice umístění, ve kterém se nachází Repository rozhraní. To se provádí specifikací Java balíčku, jenž představuje vrchol hierarchie, uvnitř které se budou Repository rozhraní vyhledávat. Kromě balíčku samotného se prochází i všechny jeho vnořené balíčky, pokud existují. Zároveň balíček jako takový nemusí být nutně specifikován konkrétně, ale může být zadán například jako regulární výraz.

Vývojáři od Spring popisují dvě možnosti jak toto provést – XML nebo Java konfigurace. Ačkoliv budou prezentovány obě varianty, jako doporučovaná a tvůrci frameworku preferovaná se uvádí Java konfigurace.

3.4.2.1 XML konfigurace

Tento způsob konfigurace se provádí přidáním elementu repositories do XML aplikačního kontextu a nastavením jeho atributu base-package na hodnotu požadovaného balíčku s Repository rozhraními. Podmínkou je, že tento aplikační kontext musí být v příslušném jmenném prostoru, který je pro každou distribuci Spring Data odlišný. Ukázka pro distribuci Spring Data JPA je znázorněna v kódu 3. Zde je jako umístění Repository rozhraní nastaven balíček com.spring.data.repository.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/data/jpa"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans.xsd
              http://www.springframework.org/schema/data/jpa
              http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

       <repositories base-package="com.spring.data.repository" />

</beans:beans>
```

Kód 3: XML konfigurace

Pokud je to zapotřebí, XML konfigurace podporuje i filtrování Repository rozhraní tak, že je možné explicitně specifikovat, které Repository do výsledku zahrnout, a které z něj naopak vyloučit. Nástrojem pro tento způsob konfigurace jsou elementy

include-filter a exclude-filter, které se zanořují přímo do elementu repositories. Konkrétní možnosti nastavení pak rozebírá dokumentace Spring Frameworku [I-22]. Ukázka vyloučení všech Repository rozhraní končících na příponu Mock je znázorněna v kódu 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://www.springframework.org/schema/data/jpa"
	xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
		http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/data/jpa
		http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
		http://www.springframework.org/schema/context
		http://www.springframework.org/schema/context/spring-context.xsd">

	<repositories base-package="com.spring.data.repository">
		<context:exclude-filter type="regex" expression=".*Mock" />
	</repositories>

</beans:beans>
```

Kód 4: XML konfigurace – filtrování

3.4.2.2 Java konfigurace

Druhou možností, jak nakonfigurovat balíček, je prostřednictvím Java konfigurace. V takovém případě v aplikaci musí existovat třída označená anotací @Configuration, která svým významem nahrazuje běžný XML aplikační kontext. Konfigurace Spring Data pak spočívá v přidání anotace ve tvaru @Enable{distribution}Repositories právě nad tuto třídu. Jak je z názvu anotace patrné, i zde je potřeba rozlišit konfiguraci podle konkrétní distribuce – ukázka pro JPA je znázorněna v kódu 5.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories("com.spring.data.repository")
@Configuration
public class ApplicationConfiguration {

	// ...

}
```

Kód 5: Java konfigurace

Obdobně jako je tomu u XML konfigurace, i v tomto případě je možné specifikovat filtrování pro jednotlivá Repository rozhraní, ačkoliv je to v tomto případě

zdlouhavější zápis. Ukázka vyloučení všech Repository rozhraní končících příponou Mock je znázorněna v kódu 6.

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.core.type.filter.RegexPatternTypeFilter;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import java.util.regex.Pattern;

@EnableJpaRepositories(
	value = "com.spring.data.repository",
	excludeFilters = {
		@ComponentScan.Filter(
			type = FilterType.CUSTOM,
			value = ApplicationConfiguration.MockRegexFilter.class
		)
	}
)
@Configuration
public class ApplicationConfiguration {

	// ...

	public static class MockRegexFilter extends RegexPatternTypeFilter {

		public MockRegexFilter() {
			super(Pattern.compile(".*Mock"));
		}

	}

}
```

Kód 6: Java konfigurace – filtrování

3.4.3 Definice obslužných rozhraní

V této kapitole budou probrány způsoby, kterými lze ve Spring Data definovat rozhraní pro přístup k datům. Toto rozhraní pak bude v aplikaci použito k obsluze konkrétních doménových objektů.

Nejjednodušším a zároveň standardním způsobem jak tohoto dosáhnout, je vytvoření vlastního, konkrétního rozhraní, které bude pomocí dědičnosti rozšiřovat Repository rozhraní ze Spring Data. V závislosti na požadované sadě základních metod je možné volit kterékoliv rozšiřující rozhraní – příkladem může být CrudRepository nebo PagingAndSortingRepository. Ukázka tohoto způsobu je zachycena v kódu 7. Třída User představuje doménovou část modelu, identifikace je datového typu Long. Je použito rozhraní typu Repository, které nedeklaruje žádné základní metody.

```
import org.springframework.data.repository.Repository;

public interface UserRepository extends Repository<User, Long> {

	// ...

}
```

Kód 7: definice obslužného Repository rozhraní

Alternativním způsobem, který vede k dosažení stejného výsledku, je použití anotace @RepositoryDefinition. Tato anotace plně nahrazuje význam Repository rozhraní a pro její použití je také vyžadováno specifikování typu doménového objektu a identifikace. Výhodou je ještě nižší míra závislosti než v případě rozhraní. To na druhou stranu může představovat nevýhodu, neboť prostřednictvím anotace není možné implicitně získat jakékoliv základní metody. Ukázka tohoto způsobu, který představuje ekvivalent k předešlému příkladu, je znázorněna v kódu 8.

```
import org.springframework.data.repository.RepositoryDefinition;

@RepositoryDefinition(domainClass = User.class, idClass = Long.class)
public interface UserRepository {

	// ...

}
```

Kód 8: alternativní definice obslužného rozhraní

Dále je možné využít anotaci @NoRepositoryBean. Ta umožňuje vyloučit rozhraní rozšiřující Repository ze správy Spring Data. Takové chování bude zpravidla použito pro definování rozšířené sady základních metod pro vlastní Repository rozhraní. Zároveň tím opět lze minimalizovat závislost na Spring Data frameworku. Ukázka definice Repository rozhraní za použití anotace @NoRepositoryBean je zobrazena v kódu 9. Ukázka obsahuje definici dvou rozhraní. Prvním je generické CreateRepository, které je anotované jako @NoRepositoryBean, a tím pádem si ho Spring Data nebude všímat. Jeho význam je však v definování základních metod, v tomto případě se jedná o CREATE metody z CrudRepository, a typu identifikace Long. Tyto specifikace pak budou brány v potaz ve všech dceřiných rozhraních, kterými se už Spring Data bude zabývat. V uvedeném případě se jedná o rozhraní UserCreateRepository. Tímto způsobem tedy lze docílit výběru pouze některých metod – zde konkrétně metod pro vytváření.

```
import org.springframework.data.repository.NoRepositoryBean;
import org.springframework.data.repository.Repository;

@NoRepositoryBean
interface CreateRepository<T> extends Repository<T, Long> {

	<S extends T> S save(S entity);

	<S extends T> Iterable<S> save(Iterable<S> entities);

	// ...

}

interface UserCreateRepository extends CreateRepository<User> {

	// ...

}
```

Kód 9: definice obslužného Repository rozhraní s výběrem základních metod

3.4.4 Použití obslužných rozhraní

Nyní již bylo vysvětleno, jak je možné nakonfigurovat umístění pro Repository rozhraní, avšak stále není objasněno, jakým způsobem se lze dostat k jejich implementacím, instancím a jakým způsobem pracovat s Repository rozhraními v místě jejich použití.

Z pohledu vývojáře bývá použití Spring Data obvykle realizováno ve čtyřech krocích. První a druhý krok je samostatně popsán v předchozích kapitolách – jde o provedení konfigurace a definování vlastních obslužných Repository rozhraní. Třetím krokem je vytvoření atributu a nastavení injektování instance Repository rozhraní. Atribut je deklarován do třídy, ve které bude příslušné Repository rozhraní použito. Nastavení injektování lze provést pomocí anotace @Autowired. Čtvrtým a zároveň posledním krokem je vlastní použití nadefinovaného Repository rozhraní. Ukázka těchto kroků pro UserRepository je znázorněna v kódu 10.

```
import org.springframework.beans.factory.annotation.Autowired;

public class SomeService {

	@Autowired
	private UserRepository userRepository;

	public void doSomething() {

		Iterable<User> users = userRepository.findAll();

		// ...

	}

}
```

Kód 10: použití Repository rozhraní

Výše popsaný způsob použití předpokládá využití Spring aplikačního kontextu k vytvoření instancí pro Repository rozhraní. Spring Data je ale možné použít i bez nutnosti vytvářet aplikační kontext. Přestože nedojde k úplnému odstranění závislostí na Spring Framework, jednotlivé instance Repository rozhraní lze vytvořit programově. K tomu poslouží třída RepositoryFactorySupport, která je specifická pro každou distribuci. Při zvolení této varianty se neprovádí standardní konfigurace, která je popsaná v předchozí části. Použití Spring Data bez aplikačního kontextu pro UserRepository je znázorněno v kódu 11.

```
import org.springframework.data.jpa.repository.support.JpaRepositoryFactory;
import org.springframework.data.repository.core.support.
	RepositoryFactorySupport;
import javax.persistence.EntityManager;

public class StandaloneService {

	private UserRepository userRepository;

	public StandaloneService(EntityManager entityManager) {

		RepositoryFactorySupport factory =
			new JpaRepositoryFactory(entityManager);

		userRepository = factory.getRepository(UserRepository.class);

	}

	// ...

}
```

Kód 11: instanciace Repository rozhraní bez aplikačního kontextu

3.4.5 Dotazovací metody

Dotazovací metody představují způsob, jakým lze prostřednictvím Spring Data deklarovat vlastní metody pro přístup k datům s tím, že není nutné manuálně definovat jejich implementaci. O to se, obdobně jako je tomu u metod z rozhraní CrudRepository nebo PagingAndSortingRepository, postará Spring Data.

V kapitole o použití Spring Data bylo vysvětleno, že za tvorbu instancí Repository rozhraní, ať už prostřednictvím aplikačního kontextu, nebo bez jeho využití, je zodpovědná třída RepositoryFactorySupport. Produktem této činnosti, nazývané také instanciace, je takzvaný proxy objekt. Jeho účelem je doplnění technologicky specifických implementací dotazů pro jednotlivé metody. Vzhledem k tomu, že pro tento proces je k dispozici pouze minimum informací – hlavička metody nadefinovaná v rozhraní, existují v zásadě dvě možnosti, jak implementaci odvodit. První je odvodit implementaci dotazu přímo z názvu metody. Druhou variantou je použití manuálně deklarovaného dotazu. O tom, kterou variantu proxy vybere, rozhoduje nastavení strategie vyhledávání dotazů. Popis těchto strategií je vysvětlen v tabulce 6, která je převzata z [I-21].

Tabulka 6: strategie vyhledávání dotazů

Strategie vyhledávání dotazu	**Popis**
CREATE	Při použití strategie CREATE se proxy objekt pokusí vytvořit technologicky specifickou implementaci dotazu na základě jména metody. Obecný přístup pak spočívá v odebrání známých předpon v názvu metody a vyhodnocení zbytého textového řetězce.
USE_DECLARED_QUERY	Strategie USE_DECLARED_QUERY znamená, že proxy objekt se pokusí nalézt deklarovanou implementaci. V případě, že se nepodaří deklaraci dotazu nalézt, proxy objekt vyhodí výjimku a proces instanciace Repository rozhraní se přeruší. Dotaz může být deklarován například pomocí anotace nebo jiným způsobem, který konkrétní Spring Data distribuce podporuje.

CREATE_IF_NOT_FOUND	Při použití CREATE_IF_NOT_FOUND lze získat výhody obou předchozích strategií. Prvním krokem je vždy vyhledávání deklarovaného dotazu, obdobně jako je tomu ve strategii USE_DECLARED_QUERY. Rozdíl je však v tom, že pokud neexistuje deklarovaný dotaz, instanciace nekončí chybou, protože implementace je doplněna pomocí názvu metody, tak jako tomu je v případě strategie CREATE.

V základním nastavení se jako výchozí strategie používá CREATE_IF_NOT_FOUND. V případě potřeby je možné strategii redefinovat. Stačí pouze rozšířit XML nebo Java konfiguraci. Nastavení strategie vyhledávání dotazů CREATE v XML konfiguraci je znázorněno v kódu 12.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://www.springframework.org/schema/data/jpa"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
		http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/data/jpa
		http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

	<repositories base-package="com.spring.data.repository"
		query-lookup-strategy="create" />

</beans:beans>
```

Kód 12: nastavení strategie vyhledávání dotazů – XML konfigurace

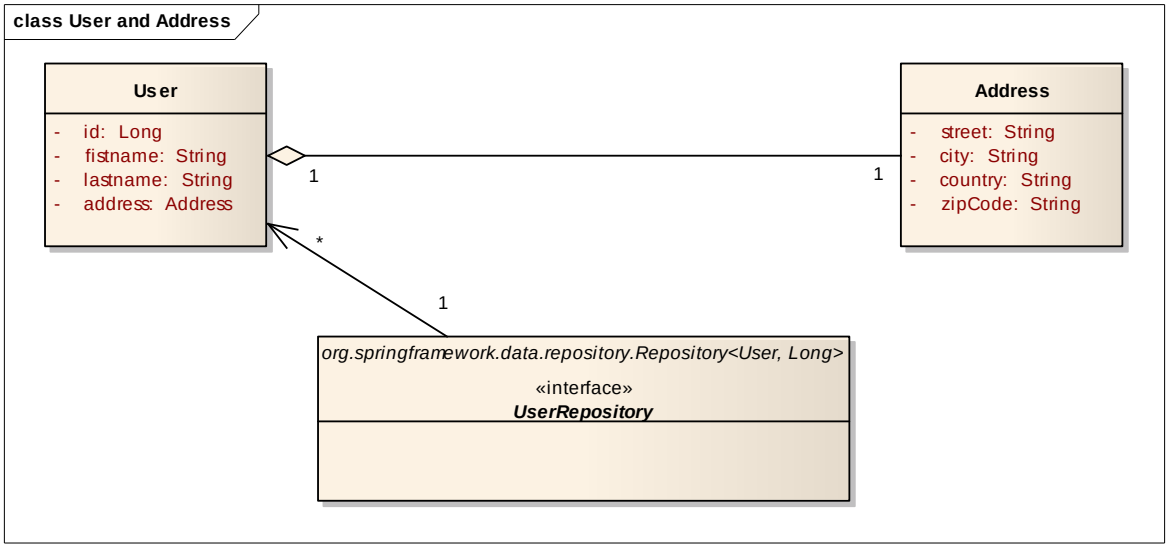
Obdobná ukázka pro Java konfiguraci je znázorněna v kódu 13.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.data.repository.query.QueryLookupStrategy.Key;

@EnableJpaRepositories(
	value = "com.spring.data.repository",
	queryLookupStrategy = Key.CREATE
)
@Configuration
public class ApplicationConfiguration {

	// ...

}
```

Kód 13: nastavení strategie vyhledávání dotazů – Java konfigurace

Nyní je již jasné, jaké jsou strategie vyhledávání dotazů a jaké chování lze od proxy objektů v tomto směru očekávat. Další částí je představení návrhu konkrétních vlastních metod. Samotný návrh musí respektovat určitá pravidla, aby bylo možné z názvu metody odvodit validní dotaz. Celý mechanizmus je založen na parsování textového řetězce, kterým je část názvu metody. Prvním krokem je odebrání předpon find...By, read...By, query...By, count...By a get...By. Druhým krokem je vlastní parsování zbytku řetězce. Úvodní část může také kromě typu operace obsahovat další výrazy. Příkladem může být findDistinct...By, který na vytvářený dotaz uplatní příznak pro odstranění duplicitních záznamů z výsledku. Po úvodní části následuje klíčové slovo By. To vystupuje jako oddělovač, který indikuje začátek výrazu, na základě kterého se budou vytvářet konkrétní kritéria dotazu. Jednotlivá kritéria lze mezi sebou oddělovat pomocí operátorů, jako jsou And a Or. Vlastní kritérium je pak tvořeno názvem atributu, případně přidruženým výrazem, který indikuje odlišné chování. Kritéria je také možné tvořit napříč hierarchií tříd.

Nyní je vhodné podívat se na ukázku definování vlastních metod, na základě kterých bude Spring Data vytvářet implementaci dotazů. Rozhraní, na kterém budou příklady ilustrovány, se nazývá UserRepository. Z pohledu vrstvy přístupu k datům se toto rozhraní stará o obsluhu doménové třídy User. Ta kromě několika základních atributů, reprezentujících identifikaci, jméno a příjmení, obsahuje vazbu na další doménovou třídu Address. Třída Address, jakožto reprezentace adresy, obsahuje pouze atributy primitivních datových typů – ulici, město, zemi a poštovní směrovací číslo. Předpokladem je, že obě třídy doménového modelu obsahují standardní get... a set... metody pro přístup a modifikaci všech svých atributů. Zjednodušený diagram tříd je znázorněn na obrázku 7.



Obrázek 7: digram tříd User, Address a UserRepository [autor]

Takto připravená infrastruktura umožňuje definovat do rozhraní UserRepository několik základních metod pro vyhledání uživatelů, které jsou znázorněny ve třech skupinách v kódu 14.

```
import org.springframework.data.repository.Repository;

public interface UserRepository extends Repository<User, Long> {

	// Single criteria

	List<User> findByFirstname(String firstname);
	List<User> findByLastname(String lastname);

	// Multiple criteria

	List<User> findByFirstnameOrLastname(String firstname, String lastname);
	List<User> findByFirstnameAndLastname(String firstname, String surname);

	// Prefix & suffix

	List<User> findDistinctByFirstname(String firstname);
	List<User> findByFirstnameIgnoreCase(String firstname);

}
```

Kód 14: rozhraní UserRepository

První skupinou jsou vyhledávací metody založené na jednom kritériu – vyhledání podle křestního jména, vyhledání podle příjmení. Druhá skupina pak demonstruje použití více kritérií oddělených spojovacími operátory – vyhledání podle jména nebo příjmení, vyhledání podle jména a příjmení. Třetí skupina pak ukazuje metody s předponami a příponami, které ovlivňují výsledek dotazu – vyhledání bez duplicitních záznamů a vyhledání ignorující velikost písmen v textu.

Další možností, kterou Spring Data při tvoření vlastních dotazů umožňuje, je navigace v hierarchii atributů. Pomocí tohoto mechanizmu je možné se v rámci dotazu odkazovat nejen na atributy dotazované třídy, ale také na atributy těchto atributů. Ukázka takového použití je znázorněna v kódu 15, kde se vyhledává uživatel podle atributů na adrese – PSČ, stát, město.

```
import org.springframework.data.repository.Repository;

public interface UserRepository extends Repository<User, Long> {

	// Property expression

	List<User> findByAddressZipCode(String zipCode);

	List<User> findByAddressCountryAndFirstname(String co, String fn);

	List<User> findByAddressCountryAndAddressCity(String co, String ci);

}
```

Kód 15: navigace v hierarchii atributů

Vlastní vyhodnocení takto definovaného kritéria spočívá v provedení jednoduchého algoritmu. Ten začíná s interpretováním dané části, např. AddressZipCode, tak, že v dotazované třídě, User, vyhledává tento řetězec jako atribut. Pokud algoritmus v tomto vyhledání uspěje, použije nalezený atribut a vyhodnocení končí. Pokud ale atribut pro dané jméno neexistuje, algoritmus začne z pravé strany k levé rozdělovat daný řetězec podle velbloudího písma na hlavní a koncovou část. Při každé iteraci zkoumá, zda v dotazované třídě existuje atribut, který by jménem odpovídal hlavní části. Pro daný příklad by algoritmus v první iteraci určil jako hlavní část AddressZip a jako koncovou část Code. Ve třídě User by však atribut jménem odpovídající této hlavní části nenalezl. Následovala by tak druhá iterace, kde by novou hlavní částí byl řetězec Address a novou koncovou částí ZipCode. Atribut address, odpovídající této hlavní části, ve třídě User existuje. Algoritmus by toto zaznamenal, vzal by koncovou část jako nový řetězec určený pro vyhodnocení a třídu Address jako novou dotazovanou třídu a pokračoval by stejným způsobem dále.

Ačkoliv tento způsob vyhodnocení hierarchie atributů ve většině případů funguje, může se v důsledku provedení popsaného algoritmu stát, že je vybrán nesprávný atribut. Pokud se vyjde z předpokladu, že třída User bude rozšířena o nový atribut s názvem addressZip tak, jak je to ukázáno v kódu 16, algoritmus by pak pro původní

příklad s řetězcem AddressZipCode, už po první iteraci vyhodnotil jako cílový atribut právě nově přidaný addressZip. Atribut addressZip je ale primitivního datového typu, proto by při vyhodnocení koncové části – řetězce Code, došlo k chybě.

```
public class User {

	//...

	private Address address;

	private String addressZip;

	// Getters & setters...

}
```

Kód 16: třída User rozšířená o atribut addressZip

Z tohoto důvodu přidali vývojáři Spring Data podporu pro manuální oddělování jednotlivých úrovní. Jako oddělovač slouží symbol podtržítka. Ukázka zápisu, který umožní explicitně oddělit jednotlivé úrovně hierarchie atributů, a tím vyřeší popsaný problém, je zobrazena v kódu 17.

```
import org.springframework.data.repository.Repository;

public interface UserRepository extends Repository<User, Long> {

	// Property expression – manual

	List<User> findByAddress_ZipCode(String zipCode);

}
```

Kód 17: manuální navigace v hierarchii atributů

Posledním bodem, který bude v souvislosti s tvorbou vlastních dotazovacích metod zmíněn, je zpracování speciálních parametrů. Tento mechanizmus zabudovaný ve Spring Data totiž umožňuje rozeznat speciální datové typy, které se vyskytují v argumentech metody nebo jsou návratovou hodnotou, a ovlivnit tak podobu dotazu a jeho výsledek. Dokumentace v tomto případě uvádí typy Pageable a Sort v kombinaci s návratovou hodnotou typu Page, pomocí kterých je možné řešit stránkování a řazení dynamicky. Jedná se o ty samé třídy, které jsou popsány v souvislosti s PagingAndSortingRepository rozhraním v úvodní části popisu Spring Data Commons. Ukázka dotazů se speciálními parametry je zobrazena v kódu 18.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.repository.Repository;

public interface UserRepository extends Repository<User, Long> {

	// Special parameter handling

	Page<User> findByFirstname(String firstname, Pageable pageable);

	List<User> findByLastname(String lastname, Pageable pageable);

	List<User> findByLastname(String lastname, Sort sort);

}
```

Kód 18: speciální parametry

První metoda z této ukázky realizuje vyhledání uživatelů podle křestního jména a parametru typu Pageable. Ten upřesňuje číslo požadované stránky, rozsah stránkování a řazení. Návratovým typem je pak generický objekt Page, který kromě toho, že obsahuje výsledek dotazu, předává dále informace o stránkování a řazení. Druhá metoda pak představuje obdobu té první, rozdílem je jednoduchý návratový typ. Třetí metoda demonstruje pouze použití řazení, a to pomocí parametru typu Sort.

3.4.6 Vlastní implementace metod

V této kapitole budou nastíněny možnosti, pomocí kterých lze zkombinovat implementace dotazovacích metod doplňovaných Spring Data s vlastním kódem. Toho bude zpravidla využito v situacích, kdy prostřednictvím Spring Data není možné nebo není žádoucí pokrýt všechny metody definované v obslužném rozhraní. Jedná se také o způsob, kterým lze přistoupit přímo ke zvolené nativní technologii. V závislosti na rozsahu použití vlastní implementace se rozlišují dvě varianty, kterými je možné požadovaného chování dosáhnout.

3.4.6.1 Vlastní implementace pro vybrané obslužné rozhraní

První variantou je přidání vlastní implementace metod pro jedno vybrané obslužné rozhraní. Toho je docíleno tak, že se nejprve definuje rozhraní pouze pro vlastní funkcionalitu a k němu je následně doplněna implementace. Druhým krokem je pak vytvoření standardního obslužného rozhraní, které však nedědí pouze ze Spring Data Repository rozhraní, ale také z rozhraní s vlastní funkcionalitou.

Ukázka je znázorněna v kódech 19, 20 a 21, kde je nejprve definováno rozhraní UserRepositoryCustom. To obsahuje deklarace metod pro vlastní funkcionalitu tj. metod, kde bude implementace doplněna explicitně.

```
public interface UserRepositoryCustom {

	void customMethod(User user);

}
```

Kód 19: rozhraní s vlastní funkcionalitou

Druhým krokem je doplnění této implementace, které je realizováno ve třídě UserRepositoryImpl. Zde je možné využít všech aspektů nativní perzistenční technologie k doplnění chování pro chybějící metody.

```
public class UserRepositoryImpl implements UserRepositoryCustom {

	@Override
	public void customMethod(User user) {

		// Custom method implementation...

	}

}
```

Kód 20: implementace rozhraní s vlastní funkcionalitou

Na závěr je definováno konečné obslužné rozhraní, které vychází z CrudRepository a UserRepositoryCustom. Obsahuje tak standardní CRUD metody ze Spring Data a zároveň metody vlastní funkcionality. Zároveň je zde opět možné doplnit vlastní dotazovací metody, jak popisuje předchozí kapitola.

```
import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, Long>,
      UserRepositoryCustom {

      // ...

}
```

Kód 21: zkombinované obslužné rozhraní

Aby představený postup fungoval bez nutnosti explicitně vytvářet instance implementujících tříd, je nutné dodržet určité jmenné konvence. Důležité je, aby třída obsahující implementace metod a zkombinované obslužné rozhraní byly ve stejném balíčku. Dále je potřeba, aby třída obsahující implementace metod byla stejně pojmenovaná jako zkombinované obslužné rozhraní s tím, že navíc obsahuje příponu. V základním nastavení se použije přípona Impl. Hodnotu je možné pomocí konfigurace předefinovat. Ukázka pro XML konfiguraci je znázorněna v kódu 22, kde je přípona tříd s vlastní implementací nastavena na hodnotu Cust. Pro předchozí příklad by potom hledaná implementace odpovídala třídě UserRepositoryCust – namísto původního UserRepositoryImpl.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.springframework.org/schema/data/jpa"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/data/jpa
            http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

      <repositories base-package="com.spring.data.repository"
            repository-impl-postfix="Cust" />

</beans:beans>
```

Kód 22: nastavení přípony tříd s vlastní implementací – XML konfigurace

Obdobný příklad jako je v předchozí ukázce s tím, že tentokrát je použita Java konfigurace, je zachycena v kódu 23.

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories(
	value = "com.spring.data.repository",
	repositoryImplementationPostfix = "Cust"
)
@Configuration
public class ApplicationConfiguration {

	// ...

}
```

Kód 23: nastavení přípony tříd s vlastní implementací – Java konfigurace

Pro tento způsob, kdy je za instanciaci třídy zodpovědný Spring Data, však existují určitá omezení. Příkladem toho je nutnost deklarace bezparametrického konstruktoru a využití anotací @Autowired k napojení instance na další objekty.

Alternativou je manuální, explicitní napojení třídy s implementací vlastní funkcionality. Toho lze dosáhnout přímo prostřednictvím aplikačního kontextu. Princip spočívá v zaregistrování třídy s vlastní implementací pod jménem, které opět vychází z názvu rozhraní a konfigurované přípony. Výhodou je možnost umístit implementaci mimo balíček s obslužnými rozhraními. Navíc lze s výslednou instancí více pracovat, je zde totiž možné použít i jiné způsoby konfigurace než pouze anotaci @Autowired. Ukázka pro XML aplikační kontext je zobrazena v kódu 24.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://www.springframework.org/schema/data/jpa"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
		http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/data/jpa
		http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

	<repositories base-package="com.spring.data.repository" />

	<beans:bean id="userRepositoryImpl" class="...">
		<!-- Further configuration... -->
	</beans:bean>

</beans:beans>
```

Kód 24: manuální napojení třídy s vlastní implementací – XML konfigurace

Obdoba příkladu pro aplikační kontext vytvořený prostřednictvím Java konfigurace je v kódu 25.

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories(
	value = "com.spring.data.repository"
)
@Configuration
public class ApplicationConfiguration {

	@Bean
	public UserRepositoryImpl userRepositoryImpl() {

		UserRepositoryImpl userRepositoryImpl = new UserRepositoryImpl();

		// Further configuration...

		return userRepositoryImpl;

	}
	// ...

}
```

Kód 25: manuální napojení třídy s vlastní implementací – Java konfigurace

3.4.6.2 Vlastní implementace pro všechna obslužná rozhraní

Druhou variantou je přidání vlastní implementace metod pro všechna obslužná rozhraní. Tento způsob v sobě zahrnuje více zásahů, avšak jeho výhodou je promítnutí implementace metody do všech obslužných rozhraní instanciovaných pomocí Spring Data. Základem je nahrazení tovární třídy, která instanci provádí, za svou vlastní. Ta pak bude produkovat instance obslužných rozhraní typovaných na rozšířené Repository rozhraní.

Prvním krokem je nadefinování rozšířeného Repository rozhraní, kam se deklarují vlastní metody. Ukázka pro JPA je zobrazena v kódu 26.

```java
import org.springframework.data.jpa.repository.JpaRepository;

public interface SharedCustomRepository<T, ID extends Serializable> extends
		JpaRepository<T, ID> {

	void sharedCustomMethod(ID identity);

}
```

Kód 26: rozšířené Repository rozhraní o vlastní metody

Druhým krokem je vytvoření implementace pro rozšiřující rozhraní. Ani zde se nelze vyhnout technologicky specifické závislosti. Tato implementace totiž bude dědit

ze základní Spring Data třídy, např. pro JPA to bude SimpleJpaRepository. Ukázka je znázorněna v kódu 27.

```
import org.springframework.data.jpa.repository.support.JpaEntityInformation;
import org.springframework.data.jpa.repository.support.SimpleJpaRepository;
import javax.persistence.EntityManager;

public class SharedCustomRepositoryImpl<T, ID extends Serializable> extends
       SimpleJpaRepository<T, ID> implements SharedCustomRepository<T, ID> {

       private EntityManager entityManager;

       public SharedCustomRepositoryImpl(JpaEntityInformation<T, ID>
                     entityInformation, EntityManager em) {
              super(entityInformation, em);
              this.entityManager = em;
       }

       public SharedCustomRepositoryImpl(Class<T> domainClass, EntityManager
                     em) {
              super(domainClass, em);
              this.entityManager = em;
       }

       @Override
       public void sharedCustomMethod(ID identity) {

              // Shared custom method implementation

       }

}
```

Kód: 27: implementace rozšířeného Repository rozhraní

Třetím krokem je vytvoření vlastní tovární třídy. Tato třída bude oproti výchozí Spring Data tovární třídě produkovat instance implementace rozšířeného rozhraní. Přístup k ní bude navíc zapouzdřen prostřednictvím standardní FactoryBean třídy ze Spring Framework. Ukázka je znázorněna v kódu 28.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.support.JpaRepositoryFactory;
import
org.springframework.data.jpa.repository.support.JpaRepositoryFactoryBean;
import org.springframework.data.repository.core.RepositoryMetadata;
import
org.springframework.data.repository.core.support.RepositoryFactorySupport;
import javax.persistence.EntityManager;

public class SharedCustomRepositoryFactoryBean<R extends JpaRepository<T, ID>,
T, ID extends Serializable> extends JpaRepositoryFactoryBean<R, T, ID> {

	protected RepositoryFactorySupport createRepositoryFactory(
			EntityManager entityManager) {

		return new SharedCustomRepositoryFactory(entityManager);

	}

	private static class SharedCustomRepositoryFactory<T, ID extends
		Serializable> extends JpaRepositoryFactory {

		private EntityManager entityManager;

		public SharedCustomRepositoryFactory(EntityManager em) {
			super(em);
			this.entityManager = em;
		}

		protected Object getTargetRepository(RepositoryMetadata rm) {

			return new SharedCustomRepositoryImpl<T, ID>(
					(Class<T>) rm.getDomainType(), entityManager);

		}

		protected Class<?> getRepositoryBaseClass(RepositoryMetadata rm) {

			return SharedCustomRepository.class;

		}

	}

}
```

Kód 28: zapouzdřená tovární třída

Čtvrtým a zároveň posledním krokem je úprava konfigurace v aplikačním kontextu. To se provede prostřednictvím výběru vlastní tovární třídy. Ukázka pro XML je znázorněna v kódu 29.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://www.springframework.org/schema/data/jpa"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
		http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/data/jpa
		http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

	<repositories base-package="com.spring.data.repository"
factory-class="com.spring.data.repository.SharedCustomRepositoryFactoryBean" />

</beans:beans>
```

Kód 29: deklarace vlastní tovární třídy – XML konfigurace

Ukázka pro Java konfiguraci je v kódu 30.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import com.spring.data.repository.SharedCustomRepositoryFactoryBean;

@EnableJpaRepositories(
	value = "com.spring.data.repository",
	repositoryFactoryBeanClass = SharedCustomRepositoryFactoryBean.class
)
@Configuration
public class ApplicationConfiguration {

	// ...

}
```

Kód 30 – deklarace vlastní tovární třídy – Java konfigurace

4 Ukázková aplikace

Cílem této části je demonstrovat již popsané možnosti frameworku Spring Data v kontextu ukázkové aplikace. Konkrétní použití technologie v sobě totiž nese nutnost vyrovnat se s omezeními, které jsou dány specifickým zadáním aplikace – jedná se především o funkční a nefunkční požadavky, jak popisuje [L-13]. Funkční požadavky definují požadovanou funkcionalitu systému, tj. co bude aplikace uživateli nabízet. Nefunkční požadavky pak odpovídají vlastnostem jako např. výkon, kapacita, dostupnost, dodržení standardů a technologií, běhové prostředí systému, atd.

Tématem ukázkové aplikace je systém pro evidenci a správu úkolů určený pro menší skupinu uživatelů. Vlastní aplikace tak zapadá do konceptu systémů pro podporu managementu, jelikož správa úkolů je jednou z podstatných činností, kterou se manažeři na všech úrovních zabývají. Aplikace bude v tomto ohledu uživateli umožňovat realizaci několika základních scénářů, které jsou pro takovéto potřeby typické.

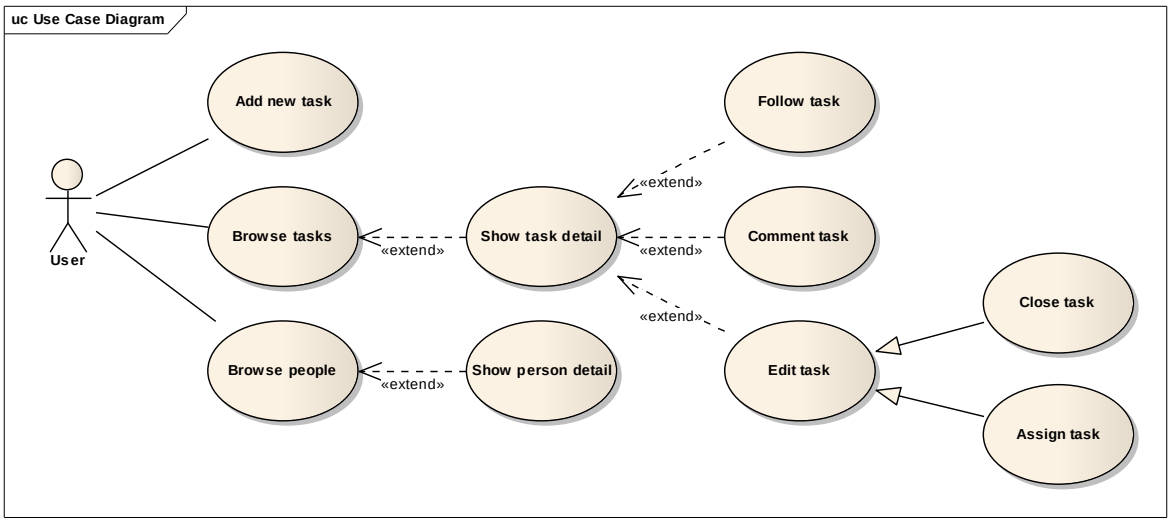
Zvolenou distribucí pro ukázkovou aplikaci je Spring Data JPA, která umožňuje vybudovat vrstvu přístupu k datům na základě standardu Java Persistence API pro práci s relačními databázemi.

4.1 Analýza

V této části budou ukázány základní modely, které popisují výslednou aplikaci. Jedná se o model typových úloh, jehož prostřednictvím se definuje základní funkcionalita, která je uživateli aplikace nabízena. Dále jde o model doménových tříd, popisující interní reprezentaci objektů, s nimiž aplikace pracuje. Na závěr bude ukázán entitně-relační model, který reprezentuje strukturu perzistovaných dat v relační databázi.

4.1.1 Model typových úloh

UML diagram typových úloh pro ukázkovou aplikaci je znázorněn na obrázku 8. Stručný slovní popis, rozepsaný pro každou typovou úlohu samostatně, je pak přiložen v tabulce 7.



Obrázek 8: diagram typových úloh ukázkové aplikace [autor]

Tabulka 7: popis případů užití ukázkové aplikace

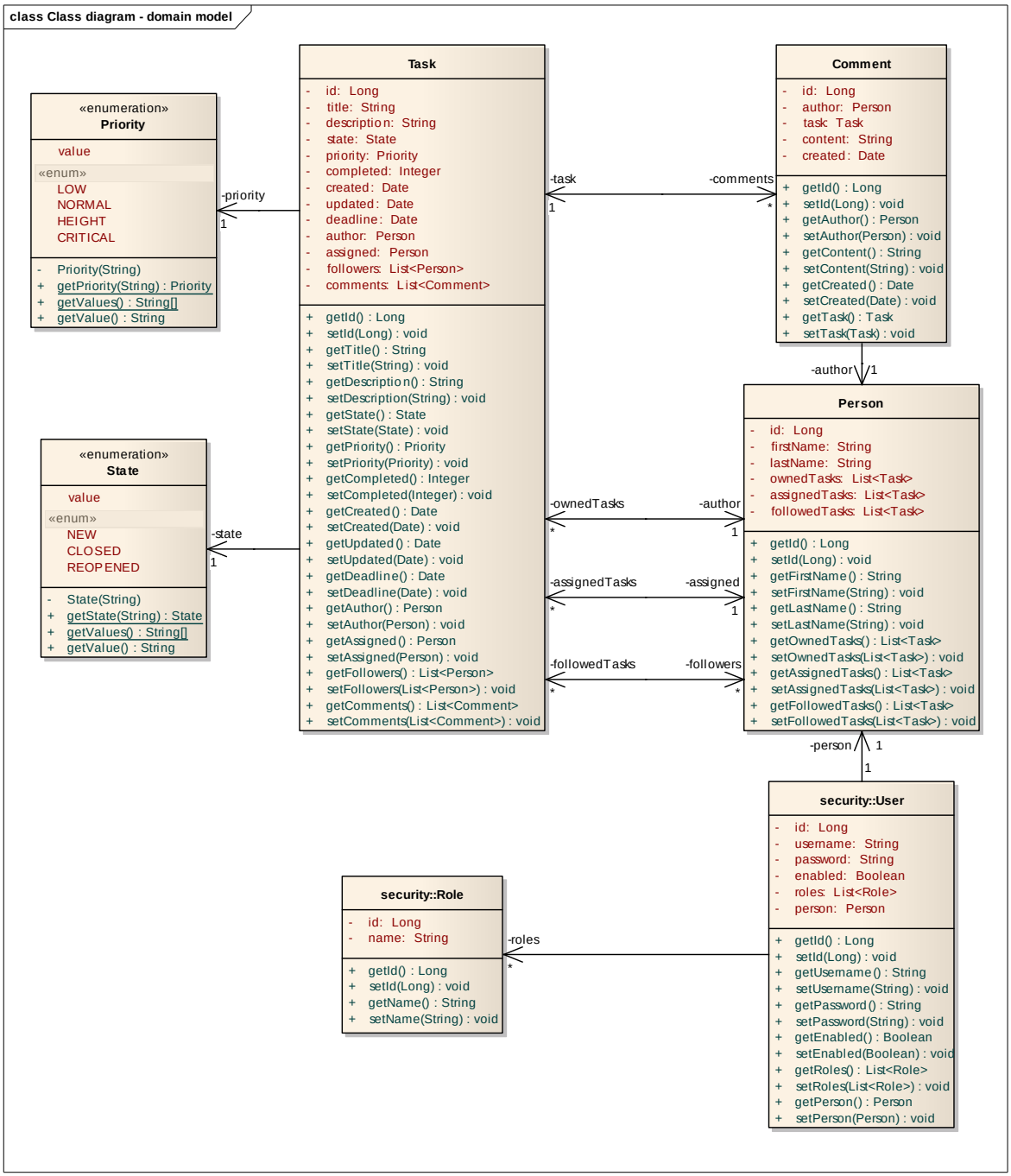
Název typové úlohy	**Popis**
Přidání nového úkolu / Add new task	V rámci založení nového úkolu uživatel iniciuje otevření příslušného formuláře. Po vyplnění všech povinných polí je možné formulář potvrdit. Pro validní vstup pak systém založí příslušný úkol a zobrazí tuto zprávu uživateli. V opačném případě systém nedovolí založení a zobrazí validační hlášku.
Procházení úkolů / Browse tasks	Uživatel bude mít možnost zobrazit si seznam všech evidovaných úkolů. Tento seznam bude možné procházet pomocí standardního stránkování. Každý záznam na stránce bude obsahovat základní informace o daném úkolu, který reprezentuje, včetně možnosti zobrazení jeho detailu.

Zobrazení detailu úkolu / Show task detail	V detailním zobrazení úkolu bude pro uživatele možné si prohlédnout všechny evidované informace k danému úkolu včetně komentářů. Zároveň se jedná o výchozí pozici pro realizaci dalších akcí jako je sledování, komentování nebo úprava úkolu.
Sledování úkolu / Follow task	Sledování umožňuje uživateli přidat / odebrat si libovolný úkol do / ze seznamu svých aktuálně sledovaných úkolů. Tento seznam je poté snadno dosažitelný a umožňuje uživateli efektivnější práci s úkoly, ke kterým není přímo přiřazen.
Komentování úkolu / Comment task	V rámci komentování úkolu může libovolný uživatel přidat textovou zprávu k danému úkolu. Tato zpráva je pak viditelná pro všechny ostatní uživatele.
Úprava úkolu / Edit task	Editace umožňuje uživateli změnit základní informace obsažené v úkolu. Součástí editace je také vložení textového popisu změny, který se zobrazí mezi komentáři.
Uzavření úkolu / Close task	Jedná se o speciální případ úpravy úkolu. Pro uzavření je nutné během editace nastavit stav úkolu na hodnotu „Uzavřený“. Uzavřený úkol je možné znovu otevřít.
Přiřazení úkolu / Assign task	Jedná se o speciální případ úpravy úkolu. Pro realizaci této úlohy je nutné během editace nastavit hodnotu pole přiřazené osoby na libovolnou zvolenou osobu. Seznam všech přiřazených úkolů se pak zobrazuje na detailu dané osoby.

Procházení osob / Browse people	Uživatel bude mít možnost zobrazit si seznam všech registrovaných osob. Tento seznam bude možné procházet pomocí standardního stránkování. Každý záznam na stránce bude obsahovat základní informace o osobě, kterou reprezentuje, včetně možnosti zobrazení jejího detailu.
Zobrazení detailu osoby / Show person detail	V detailním zobrazení osoby bude pro uživatele možné si prohlédnout všechny přiřazené úkoly pro danou osobu. Tyto úkoly budou dle stavu rozděleny na otevřené a uzavřené.

4.1.2 Doménový model

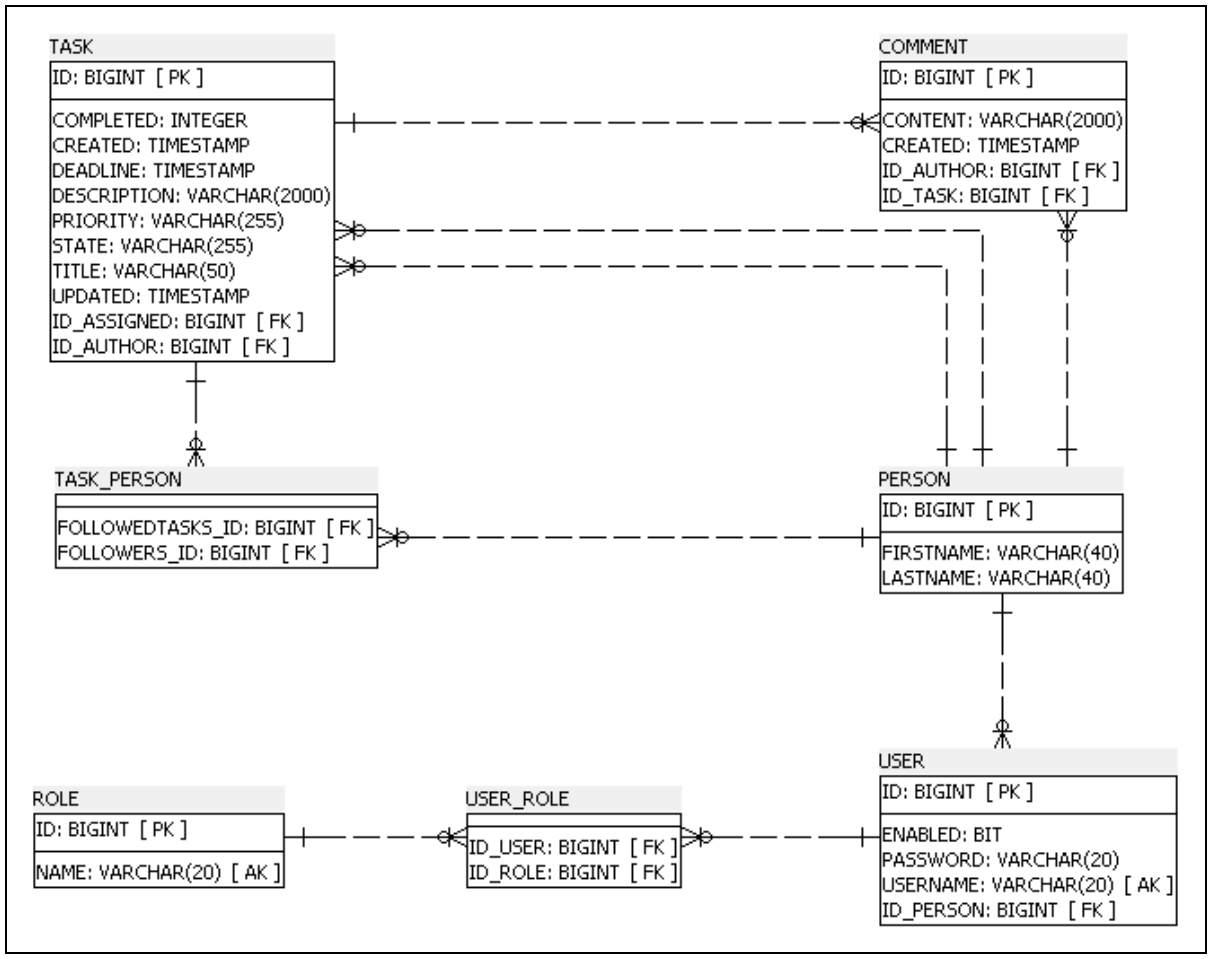
UML diagram tříd, který popisuje strukturu doménového modelu aplikace je zobrazen na obrázku 9.



Obrázek 9: diagram doménových tříd ukázkové aplikace [autor]

4.1.3 Entitně-relační model

Entitně-relační diagram, popisující strukturu dat v relační databázi aplikace, je zobrazen na obrázku 10.



Obrázek 10: ER digram ukázkové aplikace [autor]

4.2 Implementace

V této kapitole bude představena implementace ukázkové aplikace. Součástí této části bude správa projektu – a to jak vlastní aplikace, tak i databáze. Dalším bodem pak bude představení struktury aplikace. Na závěr bude rozebrána ukázka důležitých implementačních detailů z pohledu frameworku Spring Data.

4.2.1 Správa aplikace

Základní technologií, která je v ukázkové aplikaci použita pro správu projektu, je Apache Maven [I-23] (dále jen Maven). Prostřednictvím Maven je řešena správa závislostí, sestavení i spuštění aplikace. Jako Maven archetype, sloužící pro vygenerování výchozí šablony projektu, byl použit org.fluttercode.knappsack - spring-mvc-jpa-archetype na verzi 1.1. Ten, jak je i z názvu patrné, zavádí do projektu Spring MVC a JPA.

Výchozí konfigurační soubor pro Maven je pom.xml. V rámci této konfigurace jsou nastaveny veřejné repositáře, ze kterých se požadované zdroje, jako například závislosti na knihovnách nebo pluginy pro sestavení, stahují. Repositáře pro ukázkovou aplikaci jsou zobrazeny v kódu 31.

```
<repositories>
	<repository>
		<id>repository.jboss.org</id>
		<name>JBoss Repository</name>
		<url>http://repository.jboss.org/nexus/content/groups/public-
			jboss/</url>
	</repository>
	<repository>
		<id>org.springframework.maven.milestone</id>
		<name>Spring Maven Milestone Repository</name>
		<url>http://maven.springframework.org/milestone</url>
		<snapshots>
			<enabled>false</enabled>
		</snapshots>
	</repository>
</repositories>
```

Kód 31: Maven repositáře ukázkové aplikace

A nyní ke konkrétní Maven závislostem, které jsou v ukázkové aplikaci použity. Jejich základní výčet je zaznamenán v již zmíněném souboru pom.xml. Kompletní strom závislostí, vygenerovaný prostřednictvím příkazu mvn dependency:tree [I-24], je zobrazen v kódu 32.

```
cz.duspiva.lukas:tasks:war:1.0.0
+- org.springframework:spring-context:jar:3.2.1.RELEASE:compile
|  +- org.springframework:spring-beans:jar:3.2.1.RELEASE:compile
|  +- org.springframework:spring-aop:jar:3.2.1.RELEASE:compile
|  +- org.springframework:spring-expression:jar:3.2.1.RELEASE:compile
|  \- org.springframework:spring-core:jar:3.2.1.RELEASE:compile
+- org.springframework:spring-web:jar:3.2.1.RELEASE:compile
|  \- aopalliance:aopalliance:jar:1.0:compile
+- org.springframework:spring-webmvc:jar:3.2.1.RELEASE:compile
+- org.springframework:spring-orm:jar:3.2.1.RELEASE:compile
+- org.springframework:spring-test:jar:3.2.1.RELEASE:test
+- org.springframework.security:spring-security-core:jar:3.1.3.RELEASE:compile
+- org.springframework.security:spring-security-config:jar:3.1.3.RELEASE:compile
+- org.springframework.security:spring-security-web:jar:3.1.3.RELEASE:compile
|  +- org.springframework:spring-jdbc:jar:3.0.7.RELEASE:compile
|  \- org.springframework:spring-tx:jar:3.0.7.RELEASE:compile
+- org.springframework.data:spring-data-jpa:jar:1.2.0.RELEASE:compile
|  +- org.springframework.data:spring-data-commons-core:jar:1.4.0.RELEASE:compile
|  \- org.aspectj:aspectjrt:jar:1.6.12:compile
+- org.apache.tiles:tiles-core:jar:2.2.2:compile
|  +- org.apache.tiles:tiles-api:jar:2.2.2:compile
|  \- commons-digester:commons-digester:jar:2.0:compile
|     \- commons-beanutils:commons-beanutils:jar:1.8.0:compile
+- org.apache.tiles:tiles-jsp:jar:2.2.2:compile
|  +- org.apache.tiles:tiles-servlet:jar:2.2.2:compile
|  \- org.apache.tiles:tiles-template:jar:2.2.2:compile
+- org.apache.tiles:tiles-el:jar:2.2.2:compile
+- javax.servlet:servlet-api:jar:2.5:compile
+- javax.servlet.jsp:jsp-api:jar:2.2:compile
+- javax.validation:validation-api:jar:1.0.0.GA:compile
+- org.hibernate.javax.persistence:hibernate-jpa-2.0-api:jar:1.0.0.Final:compile
+- org.hibernate:hibernate-entitymanager:jar:3.6.0.Final:compile
|  +- org.hibernate:hibernate-core:jar:3.6.0.Final:compile
|  |  +- antlr:antlr:jar:2.7.6:compile
|  |  +- commons-collections:commons-collections:jar:3.1:compile
|  |  +- dom4j:dom4j:jar:1.6.1:compile
|  |  +- org.hibernate:hibernate-commons-annotations:jar:3.2.0.Final:compile
|  |  \- javax.transaction:jta:jar:1.1:compile
|  +- cglib:cglib:jar:2.2:compile
|  |  \- asm:asm:jar:3.1:compile
|  \- javassist:javassist:jar:3.12.0.GA:compile
+- org.hibernate:hibernate-validator:jar:4.2.0.Final:compile
+- commons-dbcp:commons-dbcp:jar:20030825.184428:compile
+- commons-pool:commons-pool:jar:20030825.183949:compile
+- org.hsqldb:hsqldb:jar:2.2.9:compile
+- taglibs:standard:jar:1.1.2:compile
+- log4j:log4j:jar:1.2.15:runtime
|  +- javax.mail:mail:jar:1.4:runtime
|  |  \- javax.activation:activation:jar:1.1:runtime
|  \- javax.jms:jms:jar:1.1:runtime
+- org.slf4j:slf4j-api:jar:1.5.10:compile
+- org.slf4j:jcl-over-slf4j:jar:1.5.10:runtime (scope not updated to compile)
+- org.slf4j:slf4j-log4j12:jar:1.5.10:runtime
\- junit:junit:jar:4.8.2:test
```

Kód 32: strom závislostí ukázkové aplikace

Další úlohu, kterou Maven v projektu zastává, je sestavení aplikace – tzv. build. Pro jeho manuální spuštění se používá příkaz mvn install. Ten provede spuštění příslušných Maven pluginů – explicitní konfigurace pluginů se nachází v souboru pom.xml a je zobrazena v kódu 33. Výstupem procesu je WAR soubor, určený k nasazení na webový server. WAR je vygenerován do výstupní složky target a následně zkopírovaný do lokálního repositáře Mavenu.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
	http://maven.apache.org/maven-v4_0_0.xsd">
	<build>
		<finalName>tasks</finalName>
		<plugins>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-eclipse-plugin</artifactId>
				<version>2.8</version>
				<configuration>
					<wtpversion>2.0</wtpversion>
					<downloadSources>true</downloadSources>
					<downloadJavadocs>true</downloadJavadocs>
				</configuration>
			</plugin>
			<plugin>
				<groupId>org.mortbay.jetty</groupId>
				<artifactId>maven-jetty-plugin</artifactId>
				<version>6.1.25</version>
				<configuration>
					<scanIntervalSeconds>3</scanIntervalSeconds>
				</configuration>
			</plugin>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-compiler-plugin</artifactId>
				<version>2.3.2</version>
				<configuration>
					<source>${java.version}</source>
					<target>${java.version}</target>
				</configuration>
			</plugin>
			<plugin>
				<groupId>org.codehaus.mojo</groupId>
				<artifactId>tomcat-maven-plugin</artifactId>
				<version>1.1</version>
				<configuration>
					<path>/${project.build.finalName}</path>
				</configuration>
			</plugin>
		</plugins>
	</build>
</project>
```

Kód 33: Maven konfigurace sestavení ukázkové aplikace

Třetí a zároveň poslední úloha, která zde bude v souvislosti s nástrojem Maven zmíněna, je spuštění webové aplikace na lokálním prostředí. Podmínkou úspěšnosti je v tomto ohledu provedený build a také běžící databázový server. Vlastního spuštění pak je docíleno zadáním příkazu mvn jetty:run. Tím dojde k nastartování webového serveru Jetty [I-25] na localhost a nasazení připraveného WAR souboru. Ukázková aplikace je následně dostupná na URL adrese http://localhost:8080/tasks/spring/.

4.2.2 Správa databáze

Oproti webové aplikaci, kde je správa projektu řešena sofistikovaným nástrojem, bude v případě databáze HSQL použito pouze několika základních skriptů, které jsou součástí databázové distribuce. Z pohledu kořenového adresáře databáze se tyto skripty nachází ve složce bin. Jejich popis je rozepsán v přiložené tabulce 8.

Tabulka 8: skripty pro správu databáze

Soubor	**Popis**
runDatabase.bat	Spustí databázový server na adrese localhost a portu 9001.
shutdownDatabase.bat	Zastaví běžící databázový server.
clearDatabase.bat	Vymaže databázi.
runManager.bat	Spustí jednoduché administrační rozhraní v podobě webové aplikace.
runManagerSwing.bat	Spustí jednoduché administrační rozhraní v podobě desktopové aplikace.

4.2.3 Struktura aplikace

Pohled na strukturu aplikace představuje jeden ze základních způsobů, který umožňuje pochopit jednotlivé celky, z nichž se ukázková aplikace skládá. V rámci této kapitoly bude představen pohled na organizaci zdrojového kódu, umístění konfiguračních souborů, strukturu webové části aplikace a hierarchii aplikačních kontextů používaných Spring Frameworkem.

Výchozím bodem orientace je kořenový adresář ukázkové aplikace. Ten obsahuje prvotní rozdělení souborů do dílčích podadresářů. Rozpis obsahu kořenového adresáře je přiložen v tabulce 9.

Tabulka 9: kořenový adresář ukázkové aplikace

Struktura	Popis
src/main/java	Adresář, který obsahuje hlavní zdrojové kódy Java v balíčkové struktuře. Jedná se o zdrojový kód použitý během produkčního běhu aplikaci.
src/main/resources	Adresář obsahuje vyčleněné hlavní konfigurační soubory. V tomto případě se jedná o soubory určené k produkčnímu běhu.
src/test/java	Adresář, který obsahuje testovací zdrojové kódy. Tento kód slouží k provádění automatizovaných testů.
src/test/resources	Adresář obsahuje vyčlenění testovací konfigurační soubory. V tomto případě se jedná o soubory sloužící k provádění automatizovaných testů.
src/main/webapp	Adresář obsahuje webovou část aplikace. Spadají sem prvky určené k zobrazování uživateli – stránky, styly, obrázky, atd.
pom.xml	Konfigurační soubor pro Maven – jeho význam již byl objasněn v kapitole o správě aplikace.

4.2.3.1 Zdrojové kódy

V této části bude rozebráno rozdělení zdrojových kódů ze struktury src/main/java pomocí balíčkového přístupu jazyku Java. Ten umožňuje sdružit třídy s podobnou programovou logikou do jednoho balíčku, a tím docílit přehledné a snadno udržovatelné struktury. Přehled všech balíčků a stručný popis je přiložen v tabulce 10.

Tabulka 10: popis balíčků ukázkové aplikace

Balíček	Popis
cz.duspiva.lukas.tasks.assembler	Balíček obsahuje takzvané Assembler třídy [L-14]. Jejich významem je konverze doménových objektů na DTO objekty.
cz.duspiva.lukas.tasks.controller	V tomto balíčku se nachází Controller třídy, které vycházejí ze Spring MVC. Jedná se o vrstvu aplikace, zodpovědnou za vyřízení požadavku.
cz.duspiva.lukas.tasks.domain	Tento balíček obsahuje třídy doménového modelu.
cz.duspiva.lukas.tasks.domain.security	Balíček obsahuje třídy doménového modelu, které jsou využity během autentizace a autorizace.
cz.duspiva.lukas.tasks.form.dto	V tomto balíčku jsou DTO objekty (Data Transfer Object) [L-14. Ty představují formulářovou reprezentaci doménových objektů.
cz.duspiva.lukas.tasks.repository	Balíček obsahuje Spring Data rozhraní, které tvoří vrstvu pro přístup k datům.
cz.duspiva.lukas.tasks.service	Zde jsou umístěna servisní rozhraní. Ta definují servisní metody, které pracují s doménovými objekty.
cz.duspiva.lukas.tasks.service.impl	Zde jsou umístěny implementace servisních tříd.
cz.duspiva.lukas.tasks.util	Balíček obsahuje pomocné třídy, které zapouzdřují vyčleněnou funkcionalitu statickým přístupem.
cz.duspiva.lukas.tasks.validation	V tomto balíčku jsou třídy realizující nestandardní validace.

Organizace testovacích tříd, v rámci struktury src/main/test, pak reflektuje rozdělení balíčků hlavní části zdrojového kódu.

4.2.3.2 Konfigurační soubory

Hlavní konfigurační soubory se nacházejí ve struktuře src/main/resource. Jejich výčet a popis je přiložen v tabulce č. 11.

Tabulka 11: konfigurační soubory ukázkové aplikace

Soubor	Popis
META-INF/persistence.xml	Tento soubor obsahuje konfiguraci perzistenční vrstvy JPA. Je zde definovaná tzv. perzistenční jednotka. Ta odkazuje na poskytovatele (implementaci) perzistence, jeho implicitní nastavení a v neposlední řadě také na třídy doménového modelu určené k perzistenci.
db.properties	Jde o konfiguraci přístupových údajů k databázi. Je zde definována třída ovladače, databázový dialekt, URL adresa databázového serveru a přístupové údaje uživatele.
log4j.xml	Pro logování stavu aplikace je použitý nástroj Log4j [I-26]. Jeho konfigurace umožňuje nastavit různé úrovně logování, včetně nastavení výstupů.

4.2.3.3 Webová aplikace

Poslední částí z pohledu struktury je webová aplikace, která je součástí adresářové struktury src/main/webapp. Popis pouze významných souborů a adresářů je přiložen v tabulce 12.

Tabulka 12: webová část ukázkové aplikace

Struktura	Popis
resources/css	V tomto adresáři jsou shromážděny kaskádové styly, které utvářejí grafický vzhled webové aplikace. Jako základ je použit Twitter Bootstrap verze 2.3.2 [I-27]
resources/img	Adresář obsahuje statické obrázky – například použité ikony.
resources/js	Javascriptová část webové aplikace. V tomto adresáři se nachází knihovna jQuery [I-28] a další doplňující skriptovací soubory.
WEB-INF/i18n/validation.properties	Jedná se o soubor, který obsahuje lokalizované validační hlášky.
WEB-INF/layouts/layouts.xml	Výchozí soubor frameworku Apache Tiles [I-29], který obsahuje nadefinované šablony. Na základě této konfigurace framework Tiles skládá výsledné stránky.
WEB-INF/spring	Adresář, který obsahuje XML aplikační kontexty použité pro webovou aplikaci.
WEB-INF/views	Adresář obsahuje JSP stránky v XML podobě. Jednotlivá JSP jsou dle svého zaměření seskupena do oddělených podadresářů.
WEB-INF/web.xml	Tzv. deployment descriptor – soubor obsahuje konfiguraci na úrovni servletu. Je zde registrován servlet ze Spring Framework.

Hierarchie aplikačních kontextů, kterou využívá Spring Framework jako prostředku k realizaci návrhového vzoru Dependency Injection [I-30], je rozepsána samostatně v tabulce 13.

Tabulka 13: hierarchie aplikačních kontextů ukázkové aplikace

Aplikační kontext	**Popis**
app/controllers.xml	Obsahuje definice Controller a přidružených tříd.
app/web-context.xml	Webová část aplikačního kontextu – obsahuje definice pro Spring MVC a šablonovací framework Apache Tiles.
db.xml	Kontext obsahuje definice datového zdroje, transakcí, načtení perzistenční jednotky JPA.
repository.xml	Aplikační kontext, který obsahuje definice vrstvy přístupu k datům tvořenou pomocí Spring Data.
root-context.xml	Kořenový kontext, který v sobě importuje - db.xml, repository.xml, security.xml, service.xml.
security.xml	Obsahuje definice zabezpečení z frameworku Spring Security.
service.xml	Tento kontext obsahuje definice servisních a Assembler tříd.

4.2.4 Implementace Spring Data

V této kapitole budou rozebrány implementační detaily, které souvisejí s vrstvou přístupu k datům, tvořenou pomocí frameworku Spring Data. Jelikož je v ukázkové aplikaci perzistenční vrstva řešena pomocí JPA, je zvolena odpovídající distribuce – Spring Data JPA.

4.2.4.1 Definice Repository rozhraní

Nejprve zde budou představena JPA Repository rozhraní, která jsou v ukázkové aplikaci použita během realizace nadefinovaných typových úloh. Umístění balíčku, v němž se tato rozhraní nacházejí, je dané konfigurací z aplikačního kontextu repository.xml, jak ukazuje přiložený kód 34.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://www.springframework.org/schema/data/jpa"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
http://www.springframework.org/schema/data/jpa
http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

	<repositories base-package="cz.duspiva.lukas.tasks.repository" />

</beans:beans>
```

Kód 34: konfigurace umístění Repository rozhraní ukázkové aplikace

Celkem se v aplikaci pracuje s pěti různými Repository rozhraními – jedná se CommentRepository, PersonRepository, RoleRepository, TaskRepository a UserRepository. Jmenná konvence názvu vždy odpovídá doménovému objektu, jehož obsluhu dané rozhraní realizuje. Ukázka definice pro poslední jmenované rozhraní je přiložena v kódu 35.

```
package cz.duspiva.lukas.tasks.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import cz.duspiva.lukas.tasks.domain.security.User;

public interface UserRepository extends JpaRepository<User, Long> {

	User getByUsername(String username);

	List<User> findByUsername(String username);

}
```

Kód 35: repository rozhraní UserRepository z ukázkové aplikace

Takto deklarované dotazovací metody Spring Data JPA během jejich provolání přeloží na konkrétní JPQL/HQL dotaz. Ten je následně předán k interpretaci do frameworku Hibernate, který implementuje standard JPA. Konečným výsledkem pak je SQL dotaz, který se spouští na databázi. Takovýto výstup pro metodu getByUsername z předchozího příkladu, získaný prostřednictvím logování na úrovni DEBUG, je přiložen v kódu 36.

```
DEBUG: org.hibernate.hql.ast.QueryTranslatorImpl - HQL: select generatedAlias0 from
cz.duspiva.lukas.tasks.domain.security.User as generatedAlias0 where
generatedAlias0.username=:param0
DEBUG: org.hibernate.hql.ast.QueryTranslatorImpl - SQL: select user0_.ID as ID3_,
user0_.ENABLED as ENABLED3_, user0_.PASSWORD as PASSWORD3_, user0_.ID_PERSON as ID5_3_,
user0_.USERNAME as USERNAME3_ from TASKS.USER user0_ where user0_.USERNAME=?
```

Kód 36: vygenerovaný SQL dotaz z ukázkové aplikace

Aby framework Spring Data JPA mohl úspěšně překládat dotazovací metody na SQL dotazy, je potřeba zajistit dostupnost perzistenčního kontextu JPA, aby mohlo dojít k instanciaci jednotlivých implementací rozhraní. Toho je docíleno zaregistrováním datového zdroje a tovární třídy produkující EntityManager v aplikačním kontextu db.xml – relevantní část je přiložena v kódu 37.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd">

       <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
              autowire="byName">
              <property name="jpaVendorAdapter">
                     <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                           <property name="databasePlatform"
value="${db.dialect}" />
                     </bean>
              </property>
       </bean>

       <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
              destroy-method="close" autowire="byName">
              <property name="driverClassName" value="${db.driver}" />
              <property name="url" value="${db.url}" />
              <property name="username" value="${db.username}" />
              <property name="password" value="${db.password}" />
       </bean>

       <bean
class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcess
or" />

</beans>
```

Kód 37: databázový kontext ukázkové aplikace

Konkrétní nastavení perzistenční jednotky JPA je pak zobrazeno na výřezu souboru persistance.xml – viz kód 38.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
       version="1.0">

       <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">

              <provider>org.hibernate.ejb.HibernatePersistence</provider>

              <!—Domain classes -->

              <class>cz.duspiva.lukas.tasks.domain.Comment</class>
              <class>cz.duspiva.lukas.tasks.domain.Person</class>
              <class>cz.duspiva.lukas.tasks.domain.Task</class>

              <!—Security classes -->

              <class>cz.duspiva.lukas.tasks.domain.security.User</class>
              <class>cz.duspiva.lukas.tasks.domain.security.Role</class>

       </persistence-unit>

</persistence>
```

Kód 38: nastavení perzistenční jednotky ukázkové aplikace

4.2.4.2 Transakce dotazovacích metod

Ve výchozím nastavení jsou všechny dotazovací metody Spring Data JPA transakční. Pro správné fungování je však nezbytné zaregistrovat do aplikačního kontextu implementaci transakčního manažera. Ukázka z relevantní části aplikačního kontextu db.xml je přiložena v kódu 39.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd">

       <bean id="transactionManager"
              class="org.springframework.orm.jpa.JpaTransactionManager" />

       <tx:annotation-driven />

</beans>
```

Kód 39: definice transakčního manažera ukázkové aplikace

Otevření a uzavření transakcí pak není nutné explicitně definovat v místě použití dotazovací metody. Toto je patrné například na použití metody findOne z rozhraní TaskRepository – viz zdrojový kód 40.

```
package cz.duspiva.lukas.tasks.service.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import cz.duspiva.lukas.tasks.domain.Task;
import cz.duspiva.lukas.tasks.repository.TaskRepository;
import cz.duspiva.lukas.tasks.service.TaskService;

@Service
public class TaskServiceImpl extends AbstractService implements TaskService {

	@Autowired
	private TaskRepository taskRepository;

	@Override
	public Task get(Long id) {

		//...

		return taskRepository.findOne(id);
	}


	//...

}
```

Kód 40: použití implicitně transakční metody findOne

Výpis z logu se zapnutým režimem TRACE pro transakčního manažera je přiložen v kódu 41. Před provedením SQL dotazu se otevírá transakce pro čtení – po vrácení výsledku dotazu dochází k jejímu potvrzení.

```
DEBUG: JpaTransactionManager – Found thread-bound EntityManager
[EntityManagerImpl@1c9eb5e] for JPA transaction
DEBUG: JpaTransactionManager – Creating new transaction with name
[SimpleJpaRepository.findOne]: PROPAGATION_REQUIRED,ISOLATION_DEFAULT,readOnly; ,‘
DEBUG: JDBCTransaction – begin
DEBUG: JDBCTransaction – current autocommit status: true
DEBUG: JDBCTransaction – disabling autocommit
DEBUG: JpaTransactionManager – Exposing JPA transaction as JDBC transaction
[HibernateJpaDialect$HibernateConnectionHandle@17ba1a8]
DEBUG: org.hibernate.SQL – select task0_.ID as ID2_2_, task0_.ID_ASSIGNED as ID10_2_2_,
task0_.ID_AUTHOR as ID11_2_2_, task0_.COMPLETED as COMPLETED2_2_, task0_.CREATED as
CREATED2_2_, task0_.DEADLINE as DEADLINE2_2_, task0_.DESCRIPTION as DESCRIPT5_2_2_,
task0_.PRIORITY as PRIORITY2_2_, task0_.STATE as STATE2_2_, task0_.TITLE as TITLE2_2_,
task0_.UPDATED as UPDATED2_2_, person1_.ID as ID1_0_, person1_.FIRSTNAME as FIRSTNAME1_0_,
person1_.LASTNAME as LASTNAME1_0_, person2_.ID as ID1_1_, person2_.FIRSTNAME as
FIRSTNAME1_1_, person2_.LASTNAME as LASTNAME1_1_ from TASKS.TASK task0_ inner join
TASKS.PERSON person1_ on task0_.ID_ASSIGNED=person1_.ID inner join TASKS.PERSON person2_
on task0_.ID_AUTHOR=person2_.ID where task0_.ID=?
TRACE: JpaTransactionManager – Triggering beforeCommit synchronization
TRACE: JpaTransactionManager – Triggering beforeCompletion synchronization
DEBUG: JpaTransactionManager – Initiating transaction commit
DEBUG: JpaTransactionManager – Committing JPA transaction on EntityManager
[EntityManagerImpl@1c9eb5e]
DEBUG: JDBCTransaction – commit
DEBUG: JDBCTransaction – re-enabling autocommit
DEBUG: JDBCTransaction – committed JDBC Connection
TRACE: JpaTransactionManager – Triggering afterCommit synchronization
TRACE: JpaTransactionManager – Triggering afterCompletion synchronization
DEBUG: JpaTransactionManager – Not closing pre-bound JPA EntityManager after transaction
```

Kód 41: výpis transakce z logu ukázkové aplikace

4.2.4.3 Použití Spring Data

V předchozích kapitolách bylo na použití Spring Data nahlíženo pouze jako na provolání dotazovacích metod Repository rozhraní uvnitř servisní vrstvy ukázkové aplikace. V této části budou představeny přesahy vrstvy přístupu k datům do ostatních vrstev. Konkrétně bude rozebráno použití návratových objektů v prezentační vrstvě a zpřehlednění stránkování na vrstvě Controller tříd.

V rámci architektury ukázkové aplikace se doménové objekty, získané pomocí dotazovacích metod, předávají přímo do JSP stránek – bez konverze na nezávislé DTO objekty. Aby bylo možné v prezentační vrstvě plnohodnotně využít doménových objektů, je nezbytné do souboru web.xml zaregistrovat filtr, který zajistí otevřené spojení entitního manažera s databází. Pro některé doménové objekty totiž musí být zaručen přístup do databáze – důvodem je například optimalizace načítání kolekcí. Ukázka zaregistrování příslušného filtru je přiložena v kódu 42.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	version="2.5">

	<filter>
		<!-- To bind EntityManager to the thread for the entire processing
		of the request -->
		<filter-name>openEntityManagerInViewFilter</filter-name>
		<filter-class>
		org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
		</filter-class>
	</filter>
	<filter-mapping>
	    <filter-name>openEntityManagerInViewFilter</filter-name>
	    <url-pattern>/spring/*</url-pattern>
	</filter-mapping>

</web-app>
```

Kód 42: zaregistrování openEntityManagerInViewFilter v ukázkové aplikaci

Použitím příslušného filtru je docíleno toho, že výstupy dotazovacích metod Spring Data bude možné bezpečně použít i v prezentační vrstvě. Fragmenty Spring Data se však propisují i do vrstvy Controller tříd, kde dochází k předávání informace o stránkování. Na jedné straně klient webové aplikace požaduje zobrazení určité množiny záznamů – na straně druhé server přikládá popisnou informaci k výsledkům, které vrací. Pro minimalizaci operativy, která s tímto souvisí, je v aplikačním kontextu web-context.xml zaregistrována třída PageableArgumentResolver – ukázka je přiložena v kódu 43.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

       <!-- Enables the Spring MVC @Controller programming model -->
       <annotation-driven>
       <argument-resolvers>
           <beans:bean
               class="org.springframework.data.web.PageableArgumentResolver" />
       </argument-resolvers>
       </annotation-driven>

</beans:beans>
```

Kód 43: zaregistrování PageableArgumentResolver do kontextu ukázkové aplikace

Zaregistrováním PageableArgumentResolver bude umožněno, aby obslužné metody v Controller třídách automaticky rozeznávaly argumenty související se stránkováním a překládaly je na instance Pageable. Tímto způsobem však lze rozpoznat pouze správně pojmenované argumenty. Pro zadání čísla požadované stránky je použit argument s názvem page.page. Pro zadání celkového počtu záznamů na jednu stránku pak je použito názvu page.size. Ukázka konstrukce těchto argumentů do URL adresy požadavku je zobrazena v kódu 44. Ukázka použití Pageable v obslužné metodě Controller třídy je patrná v kódu 45.

```
<li><a
href="${pageUrl}?page.page=${page.number}&#38;page.size=${page.size}">&#171;</a
></li>
```

Kód 44: argumenty stránkování v URL ukázkové aplikace

Zaregistrováním PageableArgumentResolver také dojde k aktivování anotace @PageableDefaults. Její význam spočívá v nastavení výchozího požadavku na stránkování v případě, kdy stránkování není explicitně požadována klientem. Příklad použití @PageableDefaults z ukázkové aplikace je přiložen v kódu 45.

```
package cz.duspiva.lukas.tasks.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.web.PageableDefaults;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import cz.duspiva.lukas.tasks.domain.Task;
import cz.duspiva.lukas.tasks.service.TaskService;

@RequestMapping("/task")
@Controller
public class TaskController extends AbstractController {

	private static final String KEY_TASK_PAGE = "taskPage";

	@Autowired
	private TaskService taskService;

	@RequestMapping(value = "/list")
	public String list(
		@PageableDefaults(pageNumber = 0, value = 10) Pageable pageable,
		Model model) {

		Page<Task> taskPage = taskService.getAll(pageable);

		model.addAttribute(KEY_TASK_PAGE, taskPage);

		return "task-list";
	}

	//...

}
```

Kód 45: argument stránkování v Controller třídě ukázkové aplikace

4.2.4.4 Nativní funkce JPA

Další důležitou vlastností, kterou Spring Data podporuje, je použití nativních funkcí, které daná perzistenční technologie umožňuje. V souvislosti s JPA se pak jedná především o definice vlastních dotazů v jazyce JPQL. Nicméně na fragmenty JPA lze v aplikaci narazit také během konfigurace perzistenční jednotky, a v neposlední řadě také v objektově-relačním mapování doménového modelu.

První jmenovaná možnost – definice nativního JPQL – je v ukázkové aplikaci použita pro vyhledání úkolů, které jsou v určité množině stavů a jsou přiřazeny konkrétní osobě. Výchozí nastavení strategie vyhledávání dotazů umožňuje řešit tento případ zavedením anotace @Query. Přiřazená textová hodnota pak odpovídá

konkrétnímu JPQL dotazu. Ke spárování argumentů dotazovací metody s argumenty JPQL dotazu je použito anotace @Param. Ukázka je přiložena v kódu 46.

```
package cz.duspiva.lukas.tasks.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import cz.duspiva.lukas.tasks.domain.Task;

public interface TaskRepository extends JpaRepository<Task, Long> {

	@Query(value = "select t from Task t where t.assigned = :assigned and
t.state in (:states)")
	List<Task> findByAssignedAndState(@Param("assigned") Person assigned,
			@Param("states") List<State> states);

	//...

}
```

Kód 46: použití nativního JPQL v ukázkové aplikaci

Výpis pro tento případ se zapnutým logováním na úrovni DEBUG je přiložen v kódu 47.

```
DEBUG: org.hibernate.hql.ast.QueryTranslatorImpl - HQL: select t from
cz.duspiva.lukas.tasks.domain.Task t where t.assigned = :assigned and t.state in
(:states0_, :states1_)
DEBUG: org.hibernate.hql.ast.QueryTranslatorImpl - SQL: select task0_.ID as ID2_,
task0_.ID_ASSIGNED as ID10_2_, task0_.ID_AUTHOR as ID11_2_, task0_.COMPLETED as
COMPLETED2_, task0_.CREATED as CREATED2_, task0_.DEADLINE as DEADLINE2_,
task0_.DESCRIPTION as DESCRIPT5_2_, task0_.PRIORITY as PRIORITY2_, task0_.STATE as
STATE2_, task0_.TITLE as TITLE2_, task0_.UPDATED as UPDATED2_ from TASKS.TASK task0_ where
task0_.ID_ASSIGNED=? and (task0_.STATE in (? , ?))
```

Kód 47: výpis nativního dotazu z logu ukázkové aplikace

Druhým jmenovaným bodem použití nativního JPA je konfigurace perzistenční jednotky. Jak je ukázáno na výřezu souboru persistance.xml v kódu 48, jedná se zde o možnost nastavení vlastností přímo pro implementace JPA – Hibernate. V rámci ukázkové aplikace se zde nastavuje strategie generování databázového schématu, logování SQL dotazů, transakcí, cache paměti, názvu výchozího schéma. Tato nativní konfigurace může představovat klíčovou roli během procesu optimalizace výkonu aplikace.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
       version="1.0">

       <persistence-unit name="acme" transaction-type="RESOURCE_LOCAL">

              <properties>
                     <property name="hibernate.hbm2ddl.auto" value="update" />
                     <property name="hibernate.show_sql" value="false" />
                     <property
name="hibernate.transaction.flush_before_completion" value="true" />
                     <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider" />
                     <property name="hibernate.default_schema" value="tasks" />
                     <property name="hibernate.default_batch_fetch_size"
value="3"/>
              </properties>
       </persistence-unit>

</persistence>
```

Kód 48: nastavení vlastností JPA implementace v ukázkové aplikaci

Třetím a zároveň posledním jmenovaným bodem použití nativního JPA je vlastní objektově-relační mapování doménového modelu. V rámci ukázkové aplikace je mapování ve formě JPA anotací součástí přímo doménového modelu aplikace. Mapování obsahuje informace o struktuře a vazbách relačního modelu. Ukázka mapování pro jednu ze tříd doménového modelu je přiložena v kódu 49.

```java
package cz.duspiva.lukas.tasks.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(schema = "TASKS", name = "TASK")
public class Task {

	@Id
	@GeneratedValue
	@Column(name = "ID")
	private Long id;

	@Column(name = "TITLE", nullable = false, length = 50)
	private String title;

	@Column(name = "DESCRIPTION", nullable = false, length = 2000)
	private String description;

	@Column(name = "STATE", nullable = false)
	@Enumerated(EnumType.STRING)
	private State state;

	@Column(name = "PRIORITY", nullable = false)
	@Enumerated(EnumType.STRING)
	private Priority priority;

	@Column(name = "COMPLETED", nullable = false, length = 3)
	private Integer completed;

	@Column(name = "CREATED", nullable = false)
	private Date created;

	@Column(name = "UPDATED", nullable = false)
	private Date updated;

	@Column(name = "DEADLINE", nullable = false)
	private Date deadline;

	@OneToOne
	@JoinColumn(name = "ID_AUTHOR", nullable = false)
	private Person author;


	//...

}
```

Kód 49: objektově-relační mapování ukázkové aplikace

5 Shrnutí výsledků

Na úvod této práce byla rozebrána teoretická východiska. Byly zde představeny základní pojmy jako data a databáze. Detailnější pohled byl věnován relačním a NoSQL databázím, dále pak polyglot perzistenci. Klíčovým bodem této části pak bylo vysvětlení vrstvy pro přístup k datům a prezentování základních možností její implementace prostřednictvím návrhových vzorů.

V hlavní části této práce byla popsána tvorba vrstvy pro přístup k datům pomocí frameworku Spring Data. Bylo zde rozebráno začlenění frameworku do skupiny projektů Spring, včetně představení jeho základní myšlenky a historie. Ústřední částí pak bylo popsání modulu Spring Data Commons. V tomto ohledu byla předvedena základní obslužná rozhraní, způsoby jak je definovat a používat, konfigurace prostřednictvím aplikačního kontextu a především tvorba vlastních dotazovacích metod a jejich implementací. Během vysvětlování jednotlivých možností Spring Data frameworku byl text prokládán ukázkovým zdrojovým kódem, který celkově přispěl k lepšímu pochopení objasňované problematiky.

Na závěr této práce byla předvedena ukázková aplikace, ve které byla pro realizaci jednoduchých typových úloh použita také vrstva přístupu k datům postavená na základě distribuce String Data JPA. Kromě struktury a správy projektu zde byla rozebrána především implementace spojená s frameworkem Spring Data. Bylo ukázáno, jak Spring Data z názvu dotazovací metody odvozuje JPQL dotazy, které následně JPA překládá na běžné SQL dotazy. Důraz byl také kladen na podporu transakcí a používání výstupních objektů Spring Data v prezentační vrstvě aplikace. Nechyběl zde ani popis, jakým způsobem lze přistoupit k nativním funkcím zvolené perzistenční technologie, v tomto případě JPA, pro případ optimalizace.

6 Závěry a doporučení

Tato práce si kladla za cíl představit způsob, kterým lze vytvářet vrstvu pro přístup k datům pomocí frameworku Spring Data. Během popisu obecného základu nazývaného Spring Data Commons byly postupně rozebrány jednotlivé možnosti, které framework vývojářům nabízí. Tyto možnosti byly následně společně představeny v ukázkové aplikaci, ve které bylo demonstrováno použití distribuce Spring Data JPA pro realizaci konkrétních typových úloh. Předvedením frameworku dostala práce i pohled praktického uplatnění.

Bylo zde ukázáno, že práce se Spring Data rozhraním je velmi jednoduchá a intuitivní, přestože umožňuje plnohodnotně zasahovat do mechanizmů zvolené perzistenční technologie a využívat jejího potenciálu. Na základě konkrétních příkladů by si měl čtenář, který má zkušenost s vývojem Java EE aplikací, udělat vlastní názor na tento framework a měl by s jeho pomocí být schopen implementovat vrstvu pro přístup k datům svých aplikací.

7 Seznam použité literatury

7.1 Literatura

[L-1] DATE, C. An Introduction to Database Systems. 8th ed. Boston: Pearson/Addison-Wesley, 2003, 983 s. ISBN 03-211-9784-4.

[L-2] SKLENÁK, Vilém. Data, informace, znalosti a Internet. Vyd. 1. V Praze: C.H. Beck, 2001, xvii, 507 s. C.H. Beck pro praxi. ISBN 80-717-9409-0.

[L-3] CODD, E. The relational model for database management: version 2. Reading, Mass.: Addison-Wesley, c1990, xxii, 538 p. ISBN 02-011-4192-2.

[L-4] ELMASRI, Ramez a Sham NAVATHE. Fundamentals of database systems. 6th ed. Boston: Addison-Wesley, c2011, xxvii, 1172 p. ISBN 01-360-8620-9.

[L-5] GRAND, Mark. Java Enterprise design patterns. New York: Wiley, 2001, v, 486 p. ISBN 04-713-3315-8.

[L-6] TIWARI, Shashank. Professional NoSQL. Indianapolis, Ind.: John Wiley, 2011, xxi, 361 p. Programmer to programmer.

[L-7] SADALAGE, Pramod J a Martin FOWLER. NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Upper Saddle River, NJ: Addison-Wesley, c2013, xix, 164 p. ISBN 03-218-2662-0.

[L-8] POLLACK, Mark. Spring Data: modern data access for enterprise Java. Sebastopol, CA: O'Reilly, 2013, xxi, 288 p. ISBN 14-493-2395-2.

[L-9] LAYKA, Vishal. Learn Java for Web development. xxi, 447 pages. Expert's voice in Java. ISBN 14-302-5983-3.

[L-10] HO, Clarence a Rob HARROP. Pro Spring 3. New York: Distributed to the Book trade worldwide by Springer Science Business Media, c2012, xxx, 912 p. Expert's voice in Spring. ISBN 14-302-4107-1.

[L-11] GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. Design patterns: elements of reusable object-oriented software. Massachusetts: Addison-Wesley, c1995, xv, 395 s. ISBN 078-5342633610.

[L-12] FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003, xxiv, 533 s. The Addison-Wesley Signature Series. ISBN 978-0-321-12742-6.

[L-13] ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. Vyd. 1. Překlad Bogdan Kiszka. Brno: Computer Press, 2007, 567 s. ISBN 978-80-251-1503-9.

[L-14] ALUR, Deepak, Dan MALKS a John CRUPI. Core J2EE patterns: best practices and design strategies. 2nd ed. Upper Saddle River: Prentice Hall, c2003, xxx, 650 s. ISBN 01-314-2246-4.

7.2 Internet

[I-1] FOWLER, Martin. Introduction to NoSQL by Martin Fowler: GOTO Conferences. You Tube [online]. 2013 [cit. 2014-08-16]. Dostupné z: http://youtu.be/qI_g07C_Q5I

[I-2] PIVOTAL SOFTWARE, Inc. Redis [online]. 2013 [cit. 2014-08-16]. Dostupné z: http://redis.io/

[I-3] MONGODB, Inc. MongoDB [online]. 2013 [cit. 2014-08-16]. Dostupné z: http://www.mongodb.org/

[I-4] THE APACHE SOFTWARE FOUNDATION. Apache HBase [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://hbase.apache.org/

[I-5] NEO TECHNOLOGY, Inc. Neo4j: The World's Leading Graph Database [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://www.neo4j.org/

[I-6] FOWLER, Martin. Polyglot Persistence. Martin Fowler [online]. 2011 [cit. 2014-08-16]. Dostupné z: http://martinfowler.com/bliki/PolyglotPersistence.html

[I-7] FOWLER, Martin a Pramod SADALAGE. The future is: Polyglot Persistence. Martin Fowler [online]. 2012 [cit. 2014-08-16]. Dostupné z: http://martinfowler.com/articles/nosql-intro-original.pdf

[I-8] BASHO TECHNOLOGIES, Inc. Riak [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://basho.com/riak/

[I-9] THE APACHE SOFTWARE FOUNDATION. Apache Cassandra [online]. 2009 [cit. 2014-08-16]. Dostupné z: http://cassandra.apache.org/

[I-10] Core J2EE Patterns: Data Access Object. ORACLE CORPORATION. Oracle [online]. 2001 [cit. 2014-08-16]. Dostupné z: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

[I-11] Patterns of Enterprise Application Architecture: Active Record. Martin Fowler [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://www.martinfowler.com/eaaCatalog/activeRecord.html

[I-12] JORDAN, Durran. Mongoid [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://mongoid.org/en/mongoid/

[I-13] JOHNSON, Rod. Introduction to the Spring Framework. The Server Side [online]. 2005 [cit. 2014-08-16]. Dostupné z: http://www.theserverside.com/news/1364527/Introduction-to-the-Spring-Framework

[I-14] PIVOTAL SOFTWARE, Inc. Spring [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://spring.io/projects

[I-15] PIVOTAL SOFTWARE, Inc. Open Source Software [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://gopivotal.com/oss

[I-16] THE ECLIPSE FOUNDATION. Eclipse desktop & web IDE [online]. 2014 [cit. 2014-08-16]. Dostupné z: https://www.eclipse.org/ide/

[I-17] DEMICHIEL, Linda. SUN MICROSYSTEMS. JSR-317. JSR 317: Java TM Persistence API, Version 2.0. 2008. Dostupné z: https://jcp.org/en/jsr/detail?id=317

[I-18] Origins of Spring Data Project with Neo4j. Neotechnology [online]. 2013 [cit. 2014-08-16]. Dostupné z: http://www.neotechnology.com/origins-of-spring-data-project-with-neo4j/

[I-19] PIVOTAL SOFTWARE, Inc. Spring Data [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://projects.spring.io/spring-data/

[I-20] APACHE SOFTWARE FOUNDATION. Apache Hadoop [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://hadoop.apache.org/

[I-21] POLLACK, Mark, Thomas RISBERG a Oliver GIERKE. PIVOTAL SOFTWARE, Inc. Spring Data Commons - Reference Documentation: 1.6.4.RELEASE [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://docs.spring.io/spring-data/data-commons/docs/1.6.4.RELEASE/reference/htmlsingle/

[I-22] PIVOTAL SOFTWARE, Inc. Spring Framework: The IoC container [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://docs.spring.io/spring/docs/2.5.x/reference/beans.html#beans-scanning-filters

[I-23] THE APACHE SOFTWARE FOUNDATION. Apache Maven [online]. 2002 [cit. 2014-08-16]. Dostupné z: http://maven.apache.org/

[I-24] THE APACHE SOFTWARE FOUNDATION. Maven Dependency plugin: dependency tree [online]. 2002 [cit. 2014-08-16]. Dostupné z: http://maven.apache.org/plugins/maven-dependency-plugin/tree-mojo.html

[I-25] THE ECLIPSE FOUNDATION. Jetty: Servlet Engine and Http Server [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://www.eclipse.org/jetty/

[I-26] THE APACHE SOFTWARE FOUNDATION. Apache log4j 1.2 [online]. 1999 [cit. 2014-08-16]. Dostupné z: http://logging.apache.org/log4j/1.2/

[I-27] OTTO, Mark a Jacob THORNTON. Bootstrap [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://getbootstrap.com/2.3.2

[I-28] THE JQUERY FOUNDATION. JQuery [online]. 2014 [cit. 2014-08-16]. Dostupné z: http://jquery.com/

[I-29] THE APACHE SOFTWARE FOUNDATION. Apache Tiles [online]. 2001 [cit. 2014-08-16]. Dostupné z: https://tiles.apache.org/

[I-30] FOWLER, Martin. Inversion of Control Containers and the Dependency Injection pattern. Martin Fowler [online]. 2004 [cit. 2014-08-16]. Dostupné z: http://martinfowler.com/articles/injection.html

8 Přílohy

Příloha A

Na přiloženém disku je umístěn zdrojový kód ukázkové aplikace a databáze HSQL, jejíž dostupnost je podmínkou úspěšného spuštění aplikace. Zdrojové kódy se nachází v adresáři tasks. Popis spuštění ukázkové aplikace byl zahrnut v rámci kapitoly 4.2.1 – Správa aplikace. Soubory databáze se pak nachází v adresáři hsqldb. Popis spuštění databázového serveru byl zahrnut v kapitole 4.2.2 – Správa databáze.

9 Zadání práce

FIM UHK
UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: **Lukáš Duspiva**

Obor studia: Informační management (5)
Jméno a příjmení vedoucího práce: **Filip Malý**

Název práce:
Framework Spring Data pro řešení přístupu k datům v Java Enterprise aplikacích

Název práce v AJ:
Spring Data Framework as data access solution for Java Enterprise applications

Podtitul práce:

Podtitul práce v AJ:

Cíl práce: Cílem této diplomové práce je poskytnout ucelený pohled na framework Spring Data v kontextu moderního přístupu k datům v Java Enterprise aplikacích.

Osnova práce:
Obsah
Úvod
Vrstva přístupu k datům
Framework Spring Data
Ukázková aplikace
Závěr
Seznam použitých zdrojů
přílohy

Projednáno dne: 15. 10. 2012

Podpis studenta

Podpis vedoucího práce