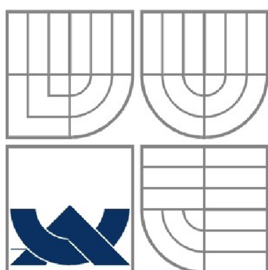
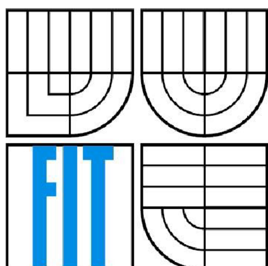


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

JAVA EE ORGANIZÉR - SOFTWAROVÁ ARCHITEKTURA

JAVA EE ORGANIZER – SOFTWARE ARCHITECTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Pavel Palát
AUTHOR

VEDOUCÍ PRÁCE Ing. Kočí Radek, Ph.D.
SUPERVISOR

BRNO 2009

!!!!! ZADANI !!!!!!

Abstrakt

Práce ukazuje základy práce s technologiemi Java EE, návrh aplikace Java EE organizér, zabývá se volbou vhodných technologii v Javě EE, zabezpečení aplikace a problematice práce se síťovými aplikacemi v Java EE technologiích. Součástí je i pojednání o zabezpečení takovýchto druhů aplikace a způsobu jejich návrhu. Primárními technologiemi používanými v této práci jsou Java EE 5 (aplikační platforma), Glassfish (aplikační server), Firebird (databáze), TopLink JPA (ORM manažer).

Abstract

This thesis deal with Java EE technologies and various subtechnologies of Java EE platform. This work is based on experience designing Java EE aplication network organizer with focus on technologies like DB connectivity (JPA, TopLink JPA), network enabled enterprise application design, remote accessing of bussiness logic in Java EE and security of the system as whole. Primary technologies used are Java EE 5 (core platform), Glassfish (application server), Firebird (database) and TopLink (ORM manager).

Klíčová slova

java, java ee, kalendář, organizace času, čas, aplikační klient

Keywords

java, java ee, calendar, tasks, todo, time organization, application client

Citace

Pavel Palát: Java EE organizér – softwarová architektura, bakalářská práce, Brno, FIT VUT v Brně, 2009

Java EE Organizér - softwarová architektura

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Radka Kočího, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Pavel Palát
20.5. 2009

Poděkování

Rád bych tímto poděkoval Ing. Kočímu, Ph.D. za pomoc při psaní práce, ostatním členům našeho řešitelského týmu (Petrovi Černému a Janu Kováři) za spolupráci a v neposlední řadě mé přítelkyni Kamile Ščurikové za trpělivost a podporu.

© Pavel Palát 2009
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah....1

1 Úvod....3

2 Analýza aplikace....4

2.1 Synchronizace kalendáře....4

2.2 Sdílení kalendáře přes síť....5

2.3 Požadavky 6

2.4 Základní architektura aplikace....6

3 Běhová a vývojová platforma....8

3.1 Aplikační server....8

3.2 Přístup k databázi – JPA....9

3.2.1 TopLink JPA....10

3.3 Databáze....11

3.3.1 Integrace Firebirdu s TopLink JPA....12

4 Architektura aplikace....14

4.1 Databáze....14

4.1.1 Entitní třídy....14

4.1.2 Perzistentní jednotka (persistence unit)....15

4.1.3 Pojmenované dotazy....16

4.2 Aplikační klient 17

4.2.1 Komunikace s aplikačním serverem....17

4.2.2 Uživatelské rozhraní....17

4.3 Byznys vrstva....18

4.3.1 Struktura byznys vrstvy....21

4.3.2 Autentizace uživatele....21

4.3.3 EJB komponenty....22

4.3.4 Přístup k byznys vrstvě z aplikačního klienta....23

4.4 Databázový model 23

4.4.1 ER diagram....24

4.4.2 Tabulky....24

5 Konfigurace software 27

5.1 Databázový server Firebird....27

5.2 Glassfish....27

5.2.1 Konfigurace připojení k databázi....27

5.2.1.1 Instalace databázového ovladače....27

5.2.1.2 Konfigurace aplikačního serveru....28
5.2.1.3 Vytvoření JNDI prostředku pro databázi....28
5.2.1.4 Konfigurace aplikace pro připojení k databázi....29
5.2.2 Konfigurace vzdáleného volání EJB....29
5.2.3 Deployment aplikace....29
5.3 Konfigurace aplikačního klienta....30
6 Závěr....32
Literatura....33
Seznam příloh....35
Příloha č. 1: Databázový model....36

1 Úvod

Organizace času a úkolů je něčím, co dnes musí dělat téměř každý. Způsobů jakým toho můžeme docílit existuje celá řada, historicky se velmi vyvíjely, od nejjednodušších poznámkových bloků, kapesních kalendářů až po složité enterprise systémy, které slouží jako organizační nástroj pro celé firmy a týmy lidí a bez nichž by si nebylo možné fungování takovýchto organizačních struktur představit.

Cílem mé práce bude seznámit se s technologiemi platformy Java Enterprise, zhodnotit možné běhové platformy (aplikační, databázové servery) a vybrat cílovou platformu pro vývoj naší aplikace (síťový kalendář postavený na modelu klient/server). Platforma Java EE byla vybrána z důvodu dobré nasaditelnosti pro tento typ aplikace, kvalitní funkcionalitě pro vývoj enterprise aplikací, včetně velkého množství dostupného SW pro ni (včetně open source). Mým cílem bude používat při vývoji pouze open source SW. Podobné důvody mě vedly i k volbě samotné platformy Java – je open source, nabízí velké množství nástrojů a knihoven (zdarma a open source), je velice vhodná pro vývoj byznys aplikací (u kterých není nejkritičtějším místem rychlost, ale stabilita robustnost aplikace, čemuž platforma Java velice napomáhá, včetně usnadnění vývoje).

Dalším cílem mé práce bude implementace základní architektury aplikace, tzn. provedení analýzy řešení, návrh datového modelu pro ni. Součástí této základní architektury bude vytvoření vývojového prostředí pro programování jednotlivých modulů – zejména vyřešení komunikace klienta a serveru, zhodnocení možností jejího zabezpečení a implementace základních funkcionalit (jádra) aplikace (jako je např. práce s uživateli). Součástí mé práce bude i práce s perzistencí – tzn. vytvoření perzistentní vrstvy pro práci s databází pomocí entitních tříd.

Další součástí práce bude vytvoření slajdů na téma Java EE a problematika bezpečnost vývoje SW na platformě Java, které by měly posloužit jako základ materiálů pro budoucí výuku.

2 Analýza aplikace

V dnešní době je velmi důležité to, abychom byli schopni kalendář sdílet a přenášet. Lidé jsou mobilní a musí si svá data nosit s sebou. Často využívají mobilní telefony, několik počítačů či notebooků a požadují, aby kalendářová data měli přístupné na kterémkoliv z těchto zařízení.

Možnosti jakým způsobem můžeme tohoto docílit je několik. V zásadě se jedná o:

- Synchronizaci zařízení (máme lokální kopii dat)
- Přístup ke kalendáři přes síť (pracujeme online)

Každý přístup má své výhody a nevýhody a hodí se na jiný případ použití.

2.1 Synchronizace kalendáře

Při synchronizaci kalendáře máme k dispozici lokální kopii dat, kterou synchronizujeme s jiným zařízením či serverem v určitém daném momentu či na požadavek uživatele.

Obecně se tento přístup může použít pro jakoukoliv topologii zařízení. Časté je použití meshové – tzn. máme jeden centrální server, který má vždy aktuální kopii dat a my se k němu připojujeme pro stažení změn. Takto např. funguje produkt Exchange [17] od firmy Microsoft , řešení postavené na groupware softwaru Funambol [16] a obecně většina groupware řešení. Často se můžeme setkat ale také s topologiemi typu peer to peer, např. při synchronizaci mobilních telefonů s počítačem, kde se nedá jasně říct, kdo je řídícím prvkem komunikace a má ty nejaktuálnější data.

Způsob jakým komunikace probíhá je závislý na konkrétních zařízeních, může se jednat o komunikaci postavenou nad protokolem TCP/IP (síťové egroupware servery, SyncML [15] přes TCP/IP kanál), ale stejně tak se může jednat o protokol používaný na sběrnici USB pro výměnu dat mezi počítačem a mobilním telefonem či protokol postavený nad technologii Bluetooth (zde se též velmi často využívá protokolu SyncML).

Výhody:

- Rychlost přístupu k datům pro uživatele – máme k dispozici lokální data a rychlost přístupu k nim závisí tudíž pouze na naší aplikaci
- Relativně malé nároky na komunikaci (objem přenesených dat a dostupnost spojení)

Nevýhody:

- Komplikovaná synchronizace v případě více zařízení, které se zapojují do procesu synchronizace
- Nutnost evidence změn které uživatel provedl od poslední synchronizace či schopnost spočítat je (může být netriviální)

- Možnost vzniku kolizí a nutnost jejich řešení (nutno počítat s tím, že určité kolize musí vyřešit uživatel)

2.2 Sdílení kalendáře přes síť

Sdílení kalendáře přes síť je podobné např. online přístupu k elektronické poště. Všechna data jsou uložena v centralizovaném úložišti, které okamžitě zpracovává všechny požadavky. Takovýto přístup tedy vyžaduje neustále připojení k serveru – nejčastěji přes Internet. Každé zařízení se vždy připojí, zadá požadavek na data, která ji následně server vrátí. Klientská aplikace může data cachovat, ale bude se jednat pouze o krátkodobou cache, která nemůže být změněna přímo klientskou aplikací. To je velký rozdíl oproti synchronizujícím kalendářům, které mají lokální data, která mohou měnit a teprve potom posílají změny na server. Při použití této metodiky přístupu (sdílení přes síť) ke kalendáři tedy nemohou vznikat kolize, resp. mohou, ale jen na straně serveru (např. dva požadavky na změnu stejné události ve stejném čase). Vznik takových kolizí je možný, i když poměrně málo pravděpodobný (nicméně musíme s nimi počítat). Řešení těchto situací je vždy odpovědnosti serveru – typická implementace jeden požadavek zpracuje a druhý zamítne.

Použití tohoto přístupu je podobné tomu, že přistupujeme ke kalendáři pomocí webového prohlížeče a používáme webové UI.

V současné době je nejčastější přístup první – tedy synchronizace. Je to dáno zejména historicky – dříve nebylo běžné mít permanentní a neomezené připojení k Internetu (zejména v mobilních telefonech) a proto bylo nutné maximálně minimalizovat toky dat a dobu připojení. Dnes je situace výrazně jiná – jsou běžné technologie přístupu k mobilnímu Internetu, které jsou zpoplatněny za přenesená data, nikoliv za strávenou dobu (např. GPRS a EDGE) a čím dál tím víc běžné je i možnost neomezeného GPRS/EDGE připojení. V domácnostech a firmách je neomezený internet (nebo alespoň neomezený pro přenosy kalendářových dat) již poměrně hodně běžný. Dá se tudíž očekávat rozmach tohoto přístupu, zejména ve spojení s applety či aplikačními klienty, které umožňují zkombinovat pohodlí práce (bez nutnosti instalace, které nabízí webové UI) s komfortem standardní desktopové aplikace.

Pro naši aplikaci jsem se rozhodl právě pro použití tohoto přístupu spolu s vytvořením aplikačního klienta, který bude přistupovat k serveru, z něhož bude vyčítat data.

Výhody:

- Nemožnost vzniku kolizí
- Máme k dispozici vždy aktuální data (pohodlné pro uživatele)
- Centralizované úložiště dat (může být i nevýhoda ve smyslu single point of failure)
- Přímočará implementace (není nutné implementovat složitou synchronizaci)

- Snazší sdílení kalendářových dat s jinými uživateli

Nevýhody:

- Nutnost trvalého připojení na Internet pro práci s kalendářem (přenášené objemy dat mohou být značné)
- Single point of failure – centralizovaný server
- Síťová komunikace může způsobovat nezanedbatelnou latenci

2.3 Požadavky

Vzhledem k naší architektuře budeme implementovat dva (více či méně) samostatné kusy software:

- Server
- Klient

Komunikace mezi nimi budeme probíhat pomocí protokolu postaveného nad TCP/IP a bude probíhat i po Internetu.

Protokol musí splňovat následující požadavky:

- Jednoduchý na implementaci
- Zabezpečení dat
  - Integrita zprávy
  - Ověření identity aplikace a serveru (je nutné počítat s možností použití kryptografie na zabezpečení síťové komunikace)
  - Správa identity uživatele:
    - Musí být zabezpečena proti pokusům o zneužití identity uživatele, musíme počítat s eventuální kompromitaci aplikace (platí zejména při použití technologie aplikačního klienta – kdy útočník může potenciálně celou aplikaci upravit či sledovat její komunikaci v nešifrované formě)
- Dostatečně úsporný

2.4 Základní architektura aplikace

Vzhledem k předpokládanému nasazení aplikace na Internet bude klientská aplikace podporovat lokální cachování dat. Tato cache bude vždy pouze krátkodobá a nebude nikdy modifikována aplikací – v případě, že chce aplikace provést změnu, musí zavolat příslušnou metodu na serveru a vyčíst si nová data (která si opět může uložit do cache). Je zde nutné počítat s tím, že data mohou být změněna i na serveru a tudíž by eventuálně měla být zajištěna automatické obnovení dat.

Všechny operace nad kalendářem musí být prováděny transakčně – tzn. buď se celá operace neprovede vůbec, nebo se provede celá.

Vzhledem k charakteru aplikace a požadavkům se jako optimální jeví třívrstvá aplikační architektura [19]:

1. Prezentační vrstva – tzn. aplikační klient, se kterým pracuje uživatel
2. Byznys vrstva – stará se o provádění jednotlivých byznys operací nad kalendářem a úkoly
3. Databáze – zajišťuje perzistentní uložení aplikačních dat

Komunikace s byznys vrstvou vždy bude probíhat pomocí síťového protokolu:

- Byznys vrstva ⇔ prezentační vrstva
  - Komunikace bude probíhat přes Internet
  - Musí být zabezpečena podle předchozích požadavků
- Byznys vrstva ⇔ databázová vrstva
  - Komunikace by musela být zabezpečena pouze v případě vzdáleného databázového serveru, u lokálního toto není žádoucí z důvodu režie, které použití šifrovaného protokolu implikuje
  - Práce s databází bude probíhat přes databázovou mezivrstvu – minimálně přes databázový ovladač

3 Běhová a vývojová platforma

Vzhledem k použitému třívrstvému modelu se jako velmi výhodná jeví technologie Java EE, která přímo podporuje vývoj byznys aplikací využívajících třívrstvý model a zároveň i podporuje vzdálený přístup k byznys vrstvě pomocí protokolů IIOP a RMI.

Java EE [13] je postavena na technologii Java, jedná se de facto o její nástavbu ve formě dalších knihoven a aplikačního serveru. Používat v ní můžeme samozřejmě všechny knihovny/funkcionality, které nabízí Java SE.

3.1 Aplikační server

Java EE aplikace jsou spouštěny v prostředí tzv. aplikačního serveru neboli kontejneru, který obsahuje několik důležitých součástí:

- Webový kontejner
  - Slouží k běhu webových aplikací na daném aplikačním serveru
  - V naší aplikaci nebude využit
- EJB kontejner
  - Obsahuje implementaci byznys vrstvy
  - Stará se o přístup k databázi
- Kontejner aplikačního klienta
  - Není přímo součástí aplikačního serveru, ale jeho infrastruktury a doprovodných nástrojů a knihoven
  - Zajišťuje možnost volání metod byznys vrstvy z prostředí aplikačního klienta

Aplikačních serverů existuje celá řada, komerčně i nekomerčně dostupných, např.:

- Glassfish – produkt firmy Sun Microsystems [05]
- RedHat Jboss [06]
- IBM Websphere[07]
- BEA WebLogic[08]
- Oracle Containers for Java EE[09]
- SAP NetWeaver[10]

Java EE jako taková je pouze soubor funkčních specifikací pro aplikační server, který musí každý server splnit aby mohl být certifikován jako Java EE kompatibilní (v příslušné verzi). Nicméně ale ani fakt, že se jedná o certifikovaný aplikační server, nezajišťuje 100% přenositelnost aplikací mezi různými aplikačními servery.

Pro naši aplikaci byl vybrán aplikační server Glassfish zejména z důvodů:

- Open source produkt

  Glassfish je open source verzí komerčního Sun Application Serveru, máme k dispozici kompletní zdrojový kód
- Je zdarma pro komerční i osobní použití
- Většinou nejrychleji drží krok s vývojem technologie Java EE
  - Často slouží jako referenční implementace pro technologie v Java EE (např. u Java Server Faces, nové specifikace Java EE 6, která je implementována částečně v novém Glassfish v3 právě pro potřeby ukázání referenční implementace)

  Použita byla verze serveru Glassfish v2, jelikož aplikace bude postavena na posledním standardu Java EE 5, který přináší výrazné vylepšení a zjednodušení vývoje oproti verzi 4 a starším. Ve vývoji je v současné době verze 6, nicméně ta zatím ještě není hotova. Dá se očekávat relativně snadná migrace na platformu Java EE 6

3.2 Přístup k databázi – JPA

Pro přístup k datům uloženým v databázi máme de facto několik možností (uvažujeme přístup z byznys vrstvy):

- Přímý přístup k databázi přes JDBC [11] (standardní Java API pro přístup k databázím, zajišťující určitou úroveň abstrakce od databázového ovladače)
- Přístup pomocí ORM mapperu (Object Relational Mapping)

ORM mapper je mechanismus, který vytváří mapování mezi relační databází a programovými objekty. O/R mapuje data mezi jednotlivými tabulkami na objekty, popř. pole objektů. Tím nám vzniká de facto objektová databáze nad klasickou relační databází.

Výhodou je usnadnění práce s databází, nemusíme vytvářet kód pro načítání dat z databáze do našich aplikačních struktur a pro zpětné ukládání těchto dat – o to se nám automaticky postará OR manažer.

Nevýhodou je o něco větší režie tohoto frameworku a o něco složitější implementace některé funkcionality, která pracuje přímo s databází, či složitější modelování některých typů relací (neplatí obecně, závisí na konkrétních OR implementacích). Nicméně obecně se dá říci, že OR představuje velice solidní technologii pro vývoj databázových aplikací.

V Javě najdeme několik implementací OR manažerů. Mezi dvě nejpoužívanější patří:

- Oracle TopLink (neboli též EclipseLink)[03]
- Hibernate[04]

Dále zde najdeme standard JPA (Java Persistence API), který je součástí oficiální specifikace Java EE 5. JPA představuje sadu API, kterou můžeme použít pro práci s OR manažerem pro

libovolnou implementaci této funkcionality (resp. JPA API). Umožňuje nám vytvářet entitní třídy (třídy, které jsou mapovány na nějakou tabulku či objekt v databázi), u kterých můžeme specifikovat mapování na konkrétní položky tabulky.

Krom standardního JPA můžeme využit i proprietárního API TopLinku či Hibernate, které nabízí další možnosti nastavení a provádění dotazů. JPA jako takové je poměrně jednoduché, složitější funkcionalita se musí řešit pomocí těchto API, ale je nepřenositelná, aplikace se potom stává závislá na této konkrétní implementaci. Pro potřeby naší aplikace jsme využívali pouze možností JPA (což ovšem bohužel neimplikuje bezproblémovou přenositelnost na jiný OR manažer z důvodů odlišného implementačního chování v některých případech).

Důležitou součástí JPA je dotazovací jazyk JPQL (Java Persistence Query Language). Dotazy tudíž nepíšeme v jazyce SQL, ale v tomto jazyku a jako výsledek dostáváme konkrétní objekty (či jejich seznamy atd.). JPQL podporuje samozřejmě použití pojmenovaných parametrů v dotazu, které bychom měli vždy používat abychom zabránili vzniku chyb typu SQL injection. Parametry mohou být libovolného typu, včetně objektů. O jejich namapování do typů používaných daným SQL serverem se postará OR manažer. Další odpovědnosti OR manažeru je samozřejmě překlad dotazů v JPQL do SQL dialektu konkrétního databázového serveru.

Součástí podpory JPQL je i možnost mít dotazy tzv. předpřipravené (prepared statements). Dotazy jsou poslány na databázový server, kde jsou zkompilovány a připraveny pro použití. Při spouštění dotazu se tudíž pouze předají parametry pro daný dotaz, čímž se výrazně ulehčí práce pro SQL server. JPA ale samozřejmě též umožňuje dotazy sestavovat dynamicky.

Pro naši aplikaci jsme zvolili implementaci ve formě Oracle TopLink JPA, jelikož se jedná o standardní OR manažer v aplikačním serveru Glassfish.

3.2.1 TopLink JPA

TopLink JPA [03] implementuje standard JPA a krom toho obsahuje též klasické TopLink API, které se používalo zejména dříve (TopLink je mnohem starší produkt než samotné JPA).

Obsahuje profily pro několik nejpoužívanějších databází, nejlepší podporu má pochopitelně pro databázový server Oracle.

TopLink z důvodu optimalizace výkonu obsahuje tzv. memory cache, do které ukládá data přečtená z databáze, tudíž dotaz vůbec nemusí vyvolat požadavek na relační databázi, jeho výsledek může být kompletně zpracován z této cache. Nevýhodou tohoto přístupu je, že pokud dojde k externí modifikaci dat (mimo rámec TopLink), tak se o této změně TopLink nemusí vůbec dozvědět a cache se může stát neaktuální. Toto chování můžeme samozřejmě vypnout, jak na úrovni celé aplikace, tak na úrovni konkrétních dotazů (dokonce pomocí standardního JPA API), nicméně potom nebude TopLink dosahovat takových výsledků jako při použití cache.

Změny do naší databáze se budou nicméně provádět vždy přes JPA API a tudíž přes TopLink a tudíž jsme cache nechali zapnutou z důvodu optimalizace výkonu.

3.3 Databáze

Volba databáze je klíčová – databáze musí být dostatečně robustní, splňovat ACID [12] kritéria (Atomicity, Consistency, Isolation, Durability) pro přístup k datům, podporovat standard SQL na patřičné úrovni a zejména musí být zajištěna interoperabilita s technologii Java EE a její implementaci v aplikačním serveru Glassfish. Požadujeme zejména:

- Dostupnost ovladače JDBC [11] (API používané pro přístup k databázím v Java)
- Kompatibilitu ovladače a databáze s technologii JPA v implementaci Oracle TopLink

Obojí splňuje poměrně velké množství existujících databázových serverů, např.:

- Oracle
- DB2
- Informix
- Sybase
- Firebird [01]
- MySQL

Pro potřeby naší aplikace jsme zvolili databázový server Firebird [01] zejména proto:

- Je open source
- Podporuje zdarma vývoj a provoz jak komerčních, tak nekomerčních aplikací
- Má k dispozici dostatečně použitelný nativní JDBC ovladač kompatibilní i s Oracle TopLink [02]

Pro provoz naší aplikace budeme předpokládat, že poběží na stejném serveru jako aplikační server, popř. s ním bude spojen bezpečnou sítí LAN. Pokud by toto nebylo splněno, tak potom by muselo dojít k zabezpečení komunikace na síťové vrstvě dalšími prostředky, možnosti samotného Firebirdu v tomto nejsou příliš velké.

Firebird může být provozován v třech typech instalací [22] [21]:

- Superserver
  - Používá multithreadovou architekturu – pro každé spojení je vytvořen nový thread (což šetří prostředky a má menší režii)
  - Cache je sdílena mezi všemi klienty
  - Je efektivní pro větší počet spojení

  - V případě pádu aplikace, chyby v UDF atd. dojde k ukončení všech spojení, to stejné platí i pro jiné problémy (porušení integrity paměti atd)
  - Lepší škálovatelnost (platí zejména pro větší počet spojení k databázi)
- Classic
  - Samostatný proces pro každé spojení
  - Spouští se z inetd / xinetd super daemonu (na UNIXových systémech)
  - Efektivní pro malý počet spojení
  - Lockování se provádí pomocí IPC
  - Cache je pro každé spojení zvlášť
- Superclassic [21]
  - Kombinuje přístup super serveru a classic architektury
    - Poměrně solidní efektivita, ale zároveň nezpůsobí pád jednoho procesu pád celého serveru
  - Používá thread pooly, které jsou vlastněny jednotlivými procesy, spouštěny jsou master procesem
    - Podobná architektura jako např. používá Apache http server v worker MPM
  - Novinka v současné době připravovaném Firebirdu 2.5
    - Má sloužit jako základ pro vylepšený threadovací model Firebirdu 3.0

Pro naši aplikaci není důležité jaký threadingový model bude v důsledku použit pro běh dané aplikace. Nicméně je nutné počítat s tím, že aplikační server Glassfish používá tzv. connection pooling – neboli má vytvořený určitý počet spojení do databáze, které udržuje neustále připojené. V momentě kdy dojde požadavek na přidělení spojení, Glassfish nejprve zkusí toto spojení přidělit z již navázaných a teprve pokud to není možné, tak naváže nové fyzické spojení. Vždy se snaží udržovat nějaké minimální množství volných spojení. Vychází to z faktu, že aplikační server často vyžaduje přítomnost databázového spojení a fyzické navázání nového spojení je poměrně hodně časově náročná operace – zejména pokud se jedná o vzdálený server.

Je tudíž běžné, že k Firebirdu bude neustále navázán poměrně velký počet spojení (podle nastavení connection pool) – tudíž může být výhodné použít threading model superserver nebo superclassic.

3.3.1 Integrace Firebirdu s TopLink JPA

TopLink JPA nepodporuje Firebird přímo, nemá pro něj vytvořený přímo profil, tudíž ho provozujeme pod profilem „generic“, který podporuje teoreticky libovolnou databázi podporující standard SQL.

Problematické se ukázalo automatické generování ID nových položek. TopLink proto podporuje několik metod [20]:

- Generátory / sekvence
- Identity položky tabulky
- Tabulka s počítadlem ID

Firebird bohužel nepodporuje syntaxi sekvencí tak, jak ji specifikuje standard a používá Oracle TopLink, tudíž se mi nepodařilo zprovoznit první metodu generování primárních klíčů. Řešením by mohlo být napsaní providera pro Firebird do TopLink a implementace podpory pro generátory tak, jak je podporuje Firebird (které mají ekvivalentní funkcionalitu, jen rozdílnou syntaxi).

My jsme se nakonec rozhodli pro použití tabulek obsahujících následující unikátní ID (tzv. table generator strategy), která funguje poměrně dobře v případech, že se neprovádí příliš mnoho insertů se stejným generátorem v jeden čas, tam funguje velice spolehlivě. V případě většího paralelismu u tohoto generátoru narůstá množství databázových kolizí. Pokud by k tomu došlo, pak by bylo výhodné implementovat podporu generátorů v Firebirdu (zde je generování atomické a spolehlivé i v rámci transakcí), nicméně to není příliš případ naší aplikace, proto jsme zvolili právě generování pomocí tabulky.

4 Architektura aplikace

Aplikace bude používat klasickou třívrstvou architekturu klasické Java EE aplikace [13].

Bude se skládat z:

- Aplikačního klienta
- Byznys vrstva
- Databáze

4.1 Databáze

Propojení databáze s aplikací je realizováno pomocí technologie JPA [14]. Rozhodl jsem se pro co nejvíce implementačně nezávislé řešení, tudíž všechny entitní třídy jsou vytvořeny pomocí anotací JPA, stejně jako všechny dotazy jsou psané v jazyce JPQL za použití objektů poskytovaných JPA. Snažili jsme se též o to nespoléhat na implementačně specifické vlastnosti (např. práce s lazy asociacemi v Hibernate a TopLink).

Databázový model jako takový – viz 4.4 Databázový model .

4.1.1 Entitní třídy

Pro každou tabulku je vytvořena jedna entitní třída, která obsahuje všechny položky dané tabulky v databáze jako proměnné dané třídy. Pomocí JPA anotací jsou svázány z položkami tabulek. Zároveň zde máme vytvořeny asociace mezi tabulkami, čímž nám kromě referencí na tabulky v databází vznikají zároveň analogické reference mezi jednotlivými třídami v JPA. U každé této reference můžeme nastavit způsob načítání:

- EAGER
  - Okamžitě načítáme krom dat dané tabulky i data pro příslušnou asociaci
  - Automaticky se provede join při sestavení dotazu, popř. JPA implementace provede další dotaz pro načtení příslušných dat
  - Tento přístup nám umožňuje vyčíst všechna data zároveň, nicméně pokud je nepotřebujeme, tak nám načítání těchto dat může prodlužovat dobu provádění a přípravy daného dotazu
- LAZY
  - Data se vyčítají až v případě potřeby

Problematické je chování LAZY asociací vzhledem k různým implementacím JPA. TopLink v případě pokusu o načtení lazy dat se nejprve pokusí použít aktuální transakci – to je ovšem možné

pouze v případě, že načítáme lazy data v byznys vrstvě v stejném kontextu jako v kterém došlo k původnímu načtení. Pokud se pokusíme o načtení dat z prezentační vrstvy, tak již původní transakce není k dispozici, proto vytvoří novou a načte data. To je poměrně pohodlné pro programátora, nicméně riskujeme načtení nekonzistentních dat, protože načítáme data mimo původní transakci. Na toto je nutné dávat pozor a počítat s tím při implementaci. Toto chování je nicméně použitelné pouze v případě lokální prezentační vrstvy, která běží na webovém kontejneru ve stejném aplikačním serveru. V případě použití remote rozhraní vzdálených EJB rozhraní a připojování se k byznys vrstvě vzdáleně (což je případ naší aplikace) není možné této vlastnosti efektivně využít, protože data entitních tříd jsou zde serializovanána a poté posílána přes síť.

Hibernate naproti tomu v případě, že se pokusíme o vyčtení lazy dat mimo session, zahlásí okamžitě chybu a data mimo session tedy nedovolí vyčíst. Vzhledem k tomu, že objem vyčítaných dat není většinou příliš velký, tak jsme většinou využívali EAGER asociace, alespoň tam, kde to dává smysl. Přístup hibernate je tedy korektnější a čistější, ale pracnější pro programátora.

Všechny entintní třídy jsou samozřejmě serializovatelné a používáme je přímo pro předávání dat prezentační vrstvě (aplikačnímu klientovi).

4.1.2 Perzistentní jednotka (persistence unit)

Perzistentní jednotka je definovaný přístup do konkrétní databáze v databázovém serveru. Je součástí konfigurace EJB aplikace v souboru persistence.xml – viz 5.2.1 Konfigurace připojení k databázi. Součástí perzistentní jednotky je konfigurace JPA provideru (v našem případě tedy TopLinku) a JNDI jméno databázového prostředku, který budeme používat. Perzistentní jednotka je tedy abstrakce, pomocí které přistupujeme do databáze.

Perzistentní jednotku můžeme referencovat pomocí resource injection a anotací @PersistenceUnit a @PersistenceContext. Resource injection spočívá v připojení anotace k proměnné dané třídy určitého pevně daného typu. Aplikační server poté provede tzv. „injection“ - neboli nastavení této proměnné na instanci implementující třídy. Tyto resource injection je možné samozřejmě provádět pouze v určitém kontextu (v našem případě v EJB). Umožňují nám snadný přístup k prostředkům aplikačního serveru bez nutnosti složitého JNDI vyhledávání (které jsme museli používat v Java EE před verzí 5).

@PersistenceUnit reprezentuje referenci na objekt typu EntityManagerFactory, který slouží jako factory třída pro vytváření objektů typu EntityManager. To je ovšem ve většině případů nepraktické nebo nežádoucí, častěji necháváme vytvoření entitního manažeru na aplikačním serveru.

K tomu nám slouží anotace @PersistenceContext, která se postará o resource injection objektu typu EntityManager, který můžeme rovnou začít používat pro operace nad JPA.

4.1.3 Pojmenované dotazy

Nad entitními třídami můžeme vytvářet pojmenované dotazy. Každý dotaz je jednoznačně identifikován svým jménem, což je řetězec libovolné délky (tedy typ String). Tento identifikátor musí být jedinečný v rámci daného EJB projektu (resp .ear archivu). Pojmenované dotazy mohou být zároveň předpřipravené, což může výrazně urychlit zpracování dotazu.

Pojmenované dotazy podporují parametry, pojmenované a indexové. Při spouštění dotazu nad entity managerem poté musíme nastavit všechny parametry dotazu a zároveň dodržet jejich typy, jinak dojde k vyvolání runtime výjimky.

Dotazy specifikujeme pomocí anotací nad danou entitní třídou (což není striktní podmínka, ale je to doporučené pro snadnější orientaci v kódu). Slouží nám k tomu anotace @NamedQueries, která obsahuje jednotlivé dotazy specifikované pomocí anotace @NamedQuery. Dotazy se píší v jazyce JPQL. Příklad:

```
@NamedQueries(
{
    @NamedQuery( name = "CalendarEntry.findByCalendarEntryId",
query = "SELECT c FROM CalendarEntry c WHERE c.calendarEntryId =
:calendarEntryId"
)
)
```

Vlastnost name specifikuje unikátní jméno pojmenovaného dotazu, query potom dotaz v jazyce JPQL. :calendarEntryId v dotazu potom říká, že se jedná o placeholder – tzn. v dotazu nám vznikl parametr podle typu CalendarEntry.calendarEntryId (field entnitní třídy CalendarEntry), který musíme nastavit při spouštění dotazu.

Dotazy jsou při deploymentu (nahrání) aplikace na aplikační server překládány a v případě, že dojde k chybě, tak je další načítání aplikace ukončeno (alespoň na aplikačním serveru Glassfish a implementaci TopLink). Ovšem pozor, kontrolována je pouze správnost JPQL dotazů, zda jsou správné výsledné dotazy v nativním SQL dialektu dané databáze kontrolováno nemusí být (záleží na implementaci JPA).

V JPA můžeme vytvářet i tzv. nativní SQL dotazy, což jsou dotazy v SQL dialektu našeho daného databázového serveru. K tomu nám analogicky slouží anotace @NamedNativeQueries a @NamedNativeQuery. Používat je můžeme naprosto analogicky, jen nám dotazy budou vracet SQL typy místo objektů instancovaných z entitních tříd.

4.2 Aplikační klient

Jedná se o samostatnou aplikaci napsanou pomocí technologie Java EE. Bude se jednat o desktopovou aplikaci provozovanou na klientském počítači s grafickým uživatelským rozhraním (GUI). Bude zprostředkovávat interakci s uživatelem. Nemusí se jednat o jedinou aplikaci, která bude s uživatelem pracovat a kterou může použít pro přístup ke kalendáři. Do budoucna by bylo možné např. doplnit webové rozhraní bez nutnosti změny byznys vrstvy – jednalo by se pouze o doprogramování další prezentační vrstvy pro naši aplikaci.

Aplikace bude pro zobrazení grafického rozhraní používat grafickou knihovnu Swing, která je součástí standardní distribuce Java VM. Aplikace tedy bude pro svůj běh vyžadovat pouze nainstalované Java VM + knihovny aplikačního serveru, které ovšem již nebude instalovat uživatel, ale budou distribuovány spolu s aplikací.

4.2.1 Komunikace s aplikačním serverem

Komunikace bude probíhat pomocí protokolu IIOP/RMI, aplikace bude vola metody v remote byznys rozhraní příslušných EJB. Nastavení spojení do aplikačního serveru nebude odpovědností samotné aplikace, ale startovacího skriptu, pomocí kterého bude aplikace spouštěna. Proces by měl být co nejvíce transparentní pro uživatele i programátora.

Komunikace může být eventuálně zabezpečena protokolem SSL, nicméně jeho použití není vyžadováno a je opět transparentní, jak pro aplikaci, tak pro uživatele. Použití SSL je doporučeno v případě použití aplikace přes nezabezpečenou síť (např. Internet), jelikož samotný IIOP/RMI nepodporuje bez SSL žádnou ochranu dat či jejich integrity a tudíž je bezpečnost takové komunikace velice malá.

4.2.2 Uživatelské rozhraní

Použita je standardní grafická UI knihovna Swing. Vzhledem k velkému množství formulářů se jeví jako výhodné využít nějaký nástroj pro generování kódu pro formuláře.

Existuje několik grafických návrhářů pro Swing, např.:

- NetBeans návrhář (Matisse) [24]
- Swing Designer [25]
- Visual Swing for Eclipse [26]

Jako nejlepší se mi jevilo použití návháře Matisse, jelikož se jedná o standardní součást vývojového prostředí NetBeans, které jsme používali pro vývoj celé aplikace. Návrhář podporuje použití všech běžných layout managerů v Javě.

Jako nejlepší pro vytváření uživatelských formulářů se jeví layout manager GroupLayout, který byl nově přidán v Javě 6, ale je možné ho použít jako knihovnu i pro starší verze Javy.

Bohužel vytvořit takto pomocí návrháře je možné jenom část formulářů, např. pro zobrazení kalendáře, denního přehledu, seznamů úkolů atd. musíme použít vlastní komponenty. Můžeme je vytvořit dvěma způsoby:

- Implementace celé vlastní komponenty (tzn. implementace paint a dalších událostí)
- Kompozice z existujících komponent (většinou vytvoření nového kontejneru)

První přístup (vlastní komponenta) je výhodný v případě, že chceme implementovat úplně novou funkcionalitu, kterou nemůžeme složit z existujících komponent. Nevýhodou je pracnost takového řešení, musíme vše implementovat sami.

Druhý způsob je o poznání jednodušší, dokonce můžeme tyto vlastní komponenty navrhovat i pomocí grafického návrháře (nebo alespoň částečně), pokud je to možné, což je ovšem pouze u statických komponent. U dynamických si příslušnou komponentu musíme naprogramovat sami ve spojení s některým layout managerem. Výběr layout managerů je zde již potom poměrně široký a můžeme použít i jednoduché layout manager (box, flow atd.) pro zobrazení jednoduchých komponent.

4.3 Byznys vrstva

Byznys vrstva slouží k provádění operací a požadavků klientů. Byznys vrstva představuje styčný bod prezentační vrstvy (v našem případě aplikačního klienta). Téměř veškerá datová logika aplikace je prováděna právě v byznys vrstvě – aplikace zadá požadavek, byznys vrstva ho zpracuje a vrátí data pro zobrazení aplikačnímu klientovi.

Klíčovou odpovědností byznys vrstvy je práce s databází – aplikační na ni nemá žádné napojení, veškerá interakce se děje právě pomocí byznys vrstvy. Používají se zde entitní třídy, jednak pro ukládání a načítání dat jednotlivých tabulek z/do databáze, ale zároveň i pro předávání dat aplikační vrstvě. Tomuto způsobu předávání je nutné ovšem věnovat pozornost, jelikož díky předávání těmto objektům nemáme zaručeno, že uživatel neprovede nějaké nepovolené změny. Např. uživatel získá seznam úkolů, u jednotlivých úkolů potom změní položku vlastník (což je samozřejmě nekorektní) a poté požádá byznys vrstvu o uložení tohoto záznamu. Máme principiálně dvě možnosti jak tyto situace řešit:

- Modifikační byznys metody budou obsahovat jako argumenty vždy jen ty parametry, které je možné u daného záznamu změnit
- Budeme provádět striktní kontrolu toho, co uživatel může změnit (tzn. musíme si, stejně jako v předchozím případě, vyčíst původní záznam z databáze)

Dá se říci, že z pohledu efektivity u běžných záznamů je jedno, který princip kontroly použijeme. U větších entitních tříd, kde je zakázáno modifikovat většinu záznamů, by bylo efektivnější použít předávání pomocí jednotlivých parametrů, ale u běžných entitních tříd je to víceméně jedno, režie je zde zanedbatelná.

Byznys vrstva je vytvořena pomocí komponent EJB (Enterprise Java Bean). EJB komponenty v Javě EE 5 a vyšší jsou velmi podobné standardním třídám (v anglické literatuře se můžeme setkat s pojmem POJO – plain old java objects), implementují se stejně, jen jsou doplněny anotacemi a vyžadují určité chování a zakazují použití určitých mechanismů. Příkladem může být např. restrikce na používání statických dat třídy – to je u EJB komponent striktně zakázané, protože podle architektury Java EE mohou EJB komponenty běžet na různých aplikačních serverech (v režimu clusteru či pouze jako failover) a může se provádět migrace klientů mezi nimi. V aplikačních serverech neexistuje žádný způsob synchronizace těchto statických dat, tudíž je použití této techniky nedoporučeno. Obecně, těchto omezení je více, s tím že některá mají charakter doporučení, jiná jsou naopak vyžadována:

- Používání reflection API
- Vracení ukazatele this či jeho jiné způsoby předávání (EJB jsou standardně spouštěny v poolu – tímto ho obcházíme a děláme méně efektivní)
- Vytváření threadů a práce s nimi (opět problematická synchronizace mezi aplikačními servery)
- Práce se sokety a obecně většina IO operací
- Práce se nativními knihovnami

Omezení je tedy celá řada, nicméně platí, že nejsou úplně striktně dodržována, můžeme si dovolit omezení ignorovat, ale musíme si být vědomi, co nám to přinese – např. to, co si můžeme dovolit v konfiguraci s jedním serverem, si nemůžeme dovolit v serverovém clusteru, kde nám klienti mohou migrovat mezi jednotlivými aplikačními servery a jejich instancemi EJB komponent. Cílem těchto omezení je udělat práci s EJB (a jejich implementaci) co nejvíce transparentní, aby se programátor nemusel zabývat detaily, jakým způsobem jsou EJB komponenty spouštěny a provozovány.

Druhy beanových komponent v EJB kontejneru:

- Session beany
  - Stateless

    Stateless beany jsou, jak už název napovídá, bezestavové – tzn. mezi jednotlivými voláními si neuchovávají žádný vnitřních stav
  - Statefull

Statefull beany si naproti tomu uchovávají vnitřní stav mezi jednotlivými voláními. Jsou mnohem náročnější na prostředky, protože aplikační server musí vytvářet mnohem více instancí než u stateless session beanů.

- Entity beany
  - Nahrazeny JPA a entitními třídami
  - Z důvodu zpětné kompatibility jsou pořád podporovány
- Message driven beany (beany řízené zprávami)
  - Beany jsou integrované s technologií JMS (Java Messaging Service), která umožňuje posílání zpráv mezi jednotlivými komponentami a kanály
  - Komunikační kanály pro mesage driven beany musí být vytvořeny v konfiguraci aplikačního serveru
  - Zprávy mohou být zasílány jak v rámci lokálního serveru, tak i přes síť.

Naši byznys vrstvu jsme vytvořili pomocí stateless beanů (jsou nejrychlejší a nejefektivnější na prostředky a plně dostačují pro vytvoření víceméně procedurální byznys vrstvě) a pomocí entitních tříd (technologie JPA). Použití dalších technologií z Java EE se pro náš projekt jevilo jako zbytečné.

Každý session bean obsahuje dvě rozhraní – lokální a vzdálené. Lokální rozhraní se používají pro volání beanů v rámci jednoho aplikačního serveru, remote rozhraní pro vzdálené volání přes síť pomocí RMI nebo IIOP. Lokální volání mají samozřejmě mnohem menší režii, protože se jedná o standardní volání metod Java objektu, pouze se zde přidává další mezivrstva (proxy na klientovi, pool v EJB kontejneru). U vzdálených (remote) volání musí docházet k serializaci, tzn. převodu jednotlivých tříd na proud bajtů a jejich následné odeslání po příslušném komunikačním kanálu (TCP/IP spojení). Na straně klienta se poté provádí opačný proces, deserializace, což má samozřejmě určitou režii, stejně jako přenos těchto dat po síti. To platí zejména v případě, že přenášíme data po nelokální síti, tam tato režie může být skutečně nezanedbatelná, což nás vede k určitému způsobu návrhu byznys vrstvy – snažit se o vyčtení co největšího počtu dat v rámci jednoho požadavku (místo použití mnoha krátkých volání) – to je po komunikačním kanálu s nezanedbatelnou latenci mnohem výhodnější. De facto platí podobná pravidla jako při práci s vzdáleným databázovým serverem.

Pro naši aplikaci jsme až na výjimky vytvářeli session beanům pouze remote rozhraní, jelikož jsou určeny pro aplikačního klienta, který může používat pouze remote rozhraní. Lokální rozhraní je pouze několik, slouží pro vzájemné propojení jednotlivých session beanů.

Pro konfiguraci rozhraní a EJB komponent jsme se snažili používat vždy anotace, které jsou v Javě EE 5 doporučené oproti původním metodám konfigurace (XML soubory), nicméně pomocí anotací není možné nakonfigurovat některé nastavení aplikace – např. nastavení spojení do databáze, které se nastavuje v konfiguračním souboru persistence.xml (resp. ukládá se zde konfigurace TopLink a JNDI jméno connection poolu)

4.3.1 Struktura byznys vrstvy

Byznys vrstvu jsem rozdělil do několika balíčků:

- cz.palat.wc.ejb
    - Obsahuje jednotlivé session beany
    - Pro jednotlivé druhy operací typicky obsahuje *Manager bean, který slouží např. pro práci s úkoly, kalendářem atd.
    - Většina beanů má pouze remote rozhraní, až na výjimky, které mají i local rozhraní
- cz.palat.wc.ejb.entity
    - Balíček obsahuje všechny entitní třídy z naší databáze (viz 4.4 Databázový model)
    - Vytvořeny striktně pomocí JPA anotací
    - Entitní třídy jsou vytvořeny včetně jejich vzájemných relací (modelujících relace mezi jednotlivými tabulkami).

4.3.2 Autentizace uživatele

Uživatelé se přihlašují prostřednictvím volání byznys vrstvy do aplikace, pro každé volání se specifikuje uživatel, s kterým pracujeme (který je aktuálně přihlášený). Nemůžeme předávat přímo informaci o uživateli – uživatelské ID jsou alokovány sekvenčně a tudíž by velice snadno mohlo dojít k jejich záměně za jiné platné uživatelské ID, což by samozřejmě snížilo bezpečnost takového systému téměř na nulu.

Z tohoto důvodu systém obsahuje koncept tzv. autentizačního tokenu, který reprezentuje mapování mezi aktuálním přihlášením a konkrétním uživatelem. Jedná se 64 bitový náhodný integer (v Javě se jedná o typ long), tudíž je odhalení by mělo být poměrně hodně netriviální operací (2^{64} možností tokenu vs např. 10-15 aktivních tokenů). Pozornost je ovšem nutné věnovat jiným typům útoků, zejména zabezpečení přenášených dat po síti, bez použití SSL či jiné podobné technologie může dojít k odposlechnutí dat či útoku man in the middle, kde může útočník získat platný autentizační token, který potom může samozřejmě použit k provedení libovolné operace pod přihlášením uživatele.

Každý autentizační token má u sebe uloženou informaci o čísle tokenu, uživatelském ID s kterým je spjat a čas posledního použití. Pokud token není využíván po nějakou dobu, tak se automaticky smaže (čímž se provede de facto odhlášení uživatele). Při každé operaci, každém volání metod byznys vrstvy se tato položka aktualizuje na aktuální datum a čas.

Proces přihlášení je potom je následující:

- Uživatel zadá uživatelské heslo a jméno

- Aplikační klient zavolá metodu byznys vrstvy pro přihlášení a předá jí tyto údaje
- Byznys vrstva vrátí buď null nebo objekt typu RemoteUser, který obsahuje číslo tokenu, který byl uživateli přiřazen
    - Číslo tokenu je generováno náhodně, ošetřuje se zde výjimka plynoucí z kolize primárních klíčů (v případě, že vygenerujeme ID tokenu, které je již v databázi použito pro jiného uživatele – což sice není příliš pravděpodobné, ale může nastat)

Jednotlivé operace potom vždy:

- Dostávají jako parametr objekt typu RemoteUser
- Pomocí beanu UserManager si tento objekt převedou na objekt typu User (aktuální uživatel) nebo zahlásí chybu. Algoritmus je následující:
    - Nejprve se smažou všechny tokeny starší než stanovený timeout
    - Operace se pokusí vyhledat daný token v databázi (tabulka USER_TOKEN)
    - Pokud ho nenajde, skončí s chybou
    - Pokud ho najde, tak aktualizuje čas posledního použití tokenu, uloží změny do databáze a vrátí objekt User (který reprezentuje už skutečného uživatele v systému)

4.3.3 EJB komponenty

CalendarManager

Komponenta slouží pro práci s událostmi v kalendáři. Je zodpovědná za jejich přidávání, mazání, změny a spočítání konkrétní množiny událostí, které se v daný den vyskytují. To není úplně triviální záležitost, jelikož zde musíme efektivně získávat seznam potenciálních událostí včetně jejich opakování (které nemůžeme mít uložené v databázi jako instance). Pro řešení tohoto problému je u každého záznamu v databázi uložena položka start date a end date (na kterých jsou vytvořeny indexy), která značí začátek prvního výskytu události a konec posledního opakování. To umožňuje velice jednoduše vybrat ty záznamy, které pro nás mohou být potenciálně relevantní pouze díky těmto dvěma položkám, což samozřejmě výrazně sníží počet položek, které musíme zpracovat.

U jednotlivých záznamů poté vždy procházíme všechny jejich opakování a zkoumá se, zda dané opakování patří do daného dne, pro které nás události zajímají. Pokud ano, tak událost přidáme do výsledného seznamu událostí (včetně konkrétního data a času u daného opakování – prezentační vrstva tedy dostane přesný seznam událostí který se den mají zobrazit).

UserManager

UserManager je zodpovědný za přihlašování a ověřování uživatelů. Obsahuje operaci pro přihlášení uživatele a alokaci nového tokenu pro něj. Obsahuje lokální rozhraní (převod tokenu na objekt User) a vzdálené (obsahující zbytek metod – např. přihlášení uživatele).
Zodpovědností tohoto EJB beanů je též práce s heslem uživatele. Do databáze se ukládá pouze jeho hash, doplněný o náhodný seed jako obrana proti rainbow table útokům. Používaným algoritmem je HMAC-SHA1 [23].

CategoryManager

EJB komponenta se stará o přidávání, mazání, změnu a získání seznamu uživatelem vytvořených kategorií. Každá položka kalendáře nebo úkol jsou vždy spjaty s konkrétní kategorií, podle které uživatel může eventuálně zobrazení událostí/úkolů vyfiltrovat.

TaskManager

Komponenta je zodpovědná za přidávání, mazání, změnu a získání seznamu uživatelem vytvořených úkolů.

4.3.4 Přístup k byznys vrstvě z aplikačního klienta

Pro přístup k byznys vrstvě musíme mít nejprve nastaveno spojení na aplikační server – viz 4.2.1 Komunikace s aplikačním serverem a 5.2.2 Konfigurace vzdáleného volání EJB. Spojení je navázáno okamžitě po startu aplikace. Pokud není možné navázat toto spojení, tak dochází k vyvolání výjimky a aplikační klient se ukončí.

Přístup k byznys vrstvě realizujeme pomocí EJB komponent. V aplikačním klientu můžeme použít klasické @EJB anotace pro resource injection komponent, ovšem s tím omezením, že se provede pouze v main třídě. Prvky navíc musí být označeny jako statické (na rozdíl od byznys vrsvy, kde prvky statické být nesmí). Jinde není resource injection v aplikačním klientu podporována.

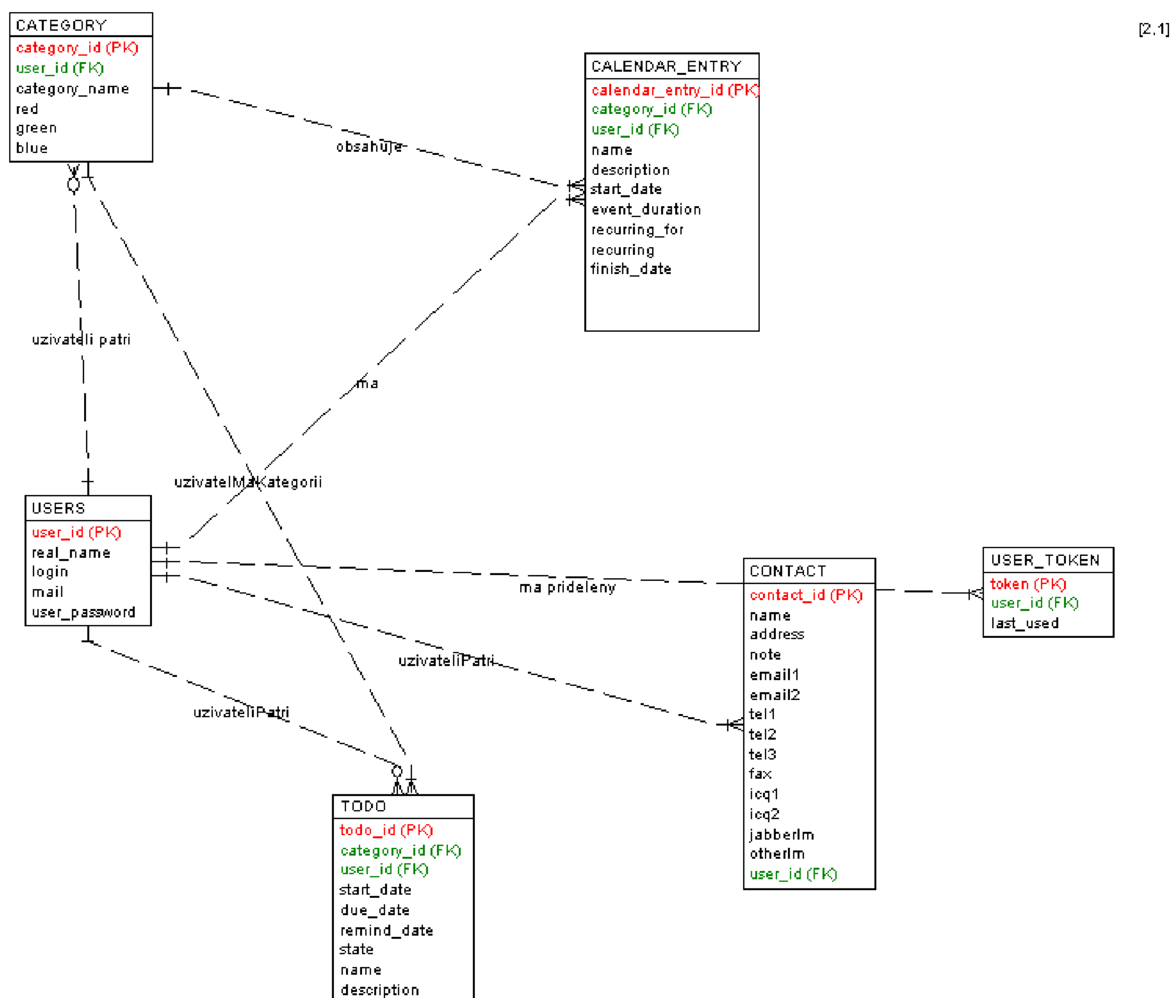
Můžeme využívat pouze vzdálených (remote) rozhraní, všechny objekty předávané přes komunikační kanál navíc musí být serializovatelné.

4.4 Databázový model

Pro naši aplikaci jsem vytvořil databázový model pro server Firebird. Pro návrh jsme využili software Toad Data Modeller pro vytvoření ER diagramu a následné vygenerování SQL skriptu pro vytvoření databáze.

Podrobnější popis databázového modelu – viz Příloha č. 1: Databázový model.

4.4.1 ER diagram



4.4.2 Tabulky

Databáze pro náš projekt byla navržena více obecně než tak, jak ji ve skutečnosti využíváme – což by mělo sloužit pro budoucí rozšiřování aplikace a doplňování dalších funkcí.

USERS

Tabulka USERS obsahuje informace o zaregistrovaných uživatelích. Položka user_pasword obsahuje hashované heslo. Konkrétní působ hashování závisí na aplikaci, databázová vrstva ho neřeší, pro ni se jedná pouze o binární data.

CATEGORY

Tabulka CATEGORY obsahuje informace o vytvořených kategoriích pro úkoly a pro schůzky (události). Každá kategorie má své jednoznačné ID a jméno, které se zobrazuje uživateli. V případě smazání kategorie je nastavena standardní kategorie u prvků, které byly původně v dané kategorii.

TODO

Obsahuje kalendářová data – seznam úkolů. Každý úkol má přiřazenou informaci o tom, kdy začíná, kdy končí a kdy má být připomenut uživateli. Úkol je vždy vytvořen nějakým konkrétním uživatelem.

CALENDAR_ENTRY

Tato tabulka slouží pro uložení údajů o schůzkách (událostech) pro daného uživatele. Dá se říci, že se jedná o nejsložitější tabulku v aplikaci, jelikož musí řešit problém opakování daných událostí.

Naše aplikace podporuje následující opakování:

- Žádné opakování
- Denní
- Týdenní
- Měsíční

U každé schůzky je stanoveno, kdy začíná, jaká je její délka v sekundách (to je nutné pro snazší počítání s opakujícími se událostmi) a datum, kdy končí poslední opakování. Pokud je událost neopakující se, tak čas konce je roven začátku události + její délce. U opakujících se událostí je datum konce rovno konci posledního opakování.

U každé události je dále možno stanovit maximální počet opakování. Informace jsou zde redundantní z důvodu optimalizace vyčítání dat z této tabulky. Vyčítání je zde zvláště důležité optimalizovat z důvodu, že se jedná o nejčastější operaci a uživatel u ní bude očekávat rychlou odezvu.

Konkrétní výpočty, které události se v daný den objevují jsou odpovědnosti byznys vrstvy. Na všech časových polích jsou z důvodu optimalizace výkonu vytvořeny indexy, což výrazně zlepšuje rychlost čtení, ale snižuje rychlost vkládání, což ale nicméně je pro naši aplikaci přijatelné, protože množství modifikací nad těmito fieldy je mnohem menší než počet čtení dat z této tabulky.

USER_TOKEN

Tabulka slouží pro ukládání o aktuálních sezeních uživatelů a jim přiděleným tokenům.

CONTACT

Tabulka obsahuje seznam kontaktů dané osoby v adresáři.

5 Konfigurace software

Konfigurace prostředí a jednotlivých komponent pro provoz naší bakalářské práce a obecně podobných softwarových aplikací postavených na technologii Java EE může být poměrně netriviální.

5.1 Databázový server Firebird

Firebird nevyžaduje žádnou speciální konfiguraci. Je pouze nutné vytvořit databázi pro náš projekt a zároveň ji nastavit uživatelská práva. Bohužel Firebird nepodporuje nastavení přístupových práv na úrovni databáze, musíme je přiřadit ručně pro každou tabulku zvlášť, nebo riskovat běh aplikace pod uživatelem SYSDBA (správce databázového serveru), což je ovšem důrazně nedoporučeno a dá se využít pouze pro vývojové účely.

5.2 Glassfish

Konfiguraci aplikačního serveru Glassfish lze rozdělit do několika částí:

- Nastavení připojení do databáze (JDBC connection pool)
- Vytvoření JNDI prostředku pro práci s ním
- Změna konfigurace v aplikaci (jméno používaného JNDI databázového prostředku)
- Konfigurace vzdáleného volání EJB

Většina těchto konfiguračních kroků je prováděna v administrační konzoly serveru Glassfish – což je webová aplikace, která je součástí aplikačního serveru. Standardně běží na portu 4848, pro přístup k ní budeme potřebovat administrátorský účet pro tuto instalaci Glassfish (měl by být vytvořený při instalaci).

5.2.1 Konfigurace připojení k databázi

5.2.1.1 Instalace databázového ovladače

Glassfish standardně neobsahuje ovladač pro databázový server Firebird, je nutné ho doinstalovat ručně. Databázový ovladač je vyvíjen jako open source projekt Jaybird [02] a můžeme ho stáhnout z oficiálních stránek Firebirdu (www.firebirdsql.org) v binární nebo zdrojové formě. Databázový driver (ve formátu archivu jar) nahrajeme do instalačního serveru s Glassfish/domains/<identifikátor domény>/lib/ext a poté nesmíme zapomenout restartovat aplikační server (což musíme udělat z příkazové řádky, webový administrátor to nepodporuje).

Poté můžeme přistoupit k dalšímu bodu konfigurace.

5.2.1.2 Konfigurace aplikačního serveru

V administrační konzoli otevřeme kartu Resources/JDBC/Connection Pools a vytvoříme nový connection pool.

Použijeme následující údaje pro nastavení:

- Typ connection poolu (Resource type): java.sql.DataSource (podpora XA transakcí je sice přítomna v Firebirdu, ale podpora Jaybirdu je špatná).
- Třída driveru (Datasource class name): org.firebirdsql.pool.FBSimpleDataSource
- Způsob kontroly spojení (Validation method): table
- Tabulka pro kontrolu spojení (Table name) : rdb$database (interní tabulka Firebirdu pro práci s metadaty serveru, je přístupná v každé databázi automaticky)
- Úroveň izolace transakcí (Transaction Isolation): read committed
- Isolation Level: Guaranted (zaškrknuto) – způsobí použití stejné úrovně izolace transakcí v každém připojení k této databázi

  Dále musíme u firebirdu vyplnit následující additional properties:
- User – uživatelské jméno pro přihlášení k databázi
- databaseName – cesta k databází, musí se jednat o absolutní cestu, aliasy u Jaybirdu nefungují kvůli nějakému bugu (tzn. např. /data/db/bp.fdb)
- Password – heslo pro přihlášení
- JDBC30DataSource – je nutné nastavit na true. Glassfish ve verzi 2u1 a vyšší totiž předpokládá podporu JDBC40 vlastností, které jsou v Jaybirdu implementovány jako stub funkce a tudíž nefunkční. Touto volbou vnutíme použití pouze těch funkcí, které specifikuje standard JDBC verze 3. Bez této volby nebude přístup do databáze pracovat korektně (a bohužel se chyba bude projevovat pouze v některých případech)!

Po dokončení těchto nastavení můžeme kliknout na tlačítko ping a otestovat funkčnost spojení do databáze. Pokud se nám neobjeví hláška informující o úspěchu dané operace, nemůžeme pokračovat dále a musíme problém nejprve odstranit.

5.2.1.3 Vytvoření JNDI prostředku pro databázi

V administrační konzoli otevřeme kartu Resources/JDBC/JDBC resources a vytvoříme nový prostředek. Vyplníme jeho JNDI jméno a vybereme náš connection pool, který jsme vytvořili v předchozím kroku a přidáme tento prostředek. JNDI je obecná adresářová technologie, která umožňuje aplikacím vyhledat prostředek aplikačního serveru.

Aplikace se k databázi připojují právě pomocí JNDI prostředků (resp. pomocí technologie JNDI si daný prostředek vyhledají). O správu spojení se stará samotný aplikační server.

5.2.1.4 Konfigurace aplikace pro připojení k databázi

Aby se aplikace mohla úspěšně připojit k databázi, musíme ji nastavit, k jakému JNDI prostředku se má připojovat. Tato konfigurace je uložena v souboru persistence.xml v projektu WebCalendar-EJB. Soubor má následující strukturu:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
 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">
     <persistence-unit   name="WebCalendar-EJBPU"   transaction-
type="JTA">
    <jta-data-source>WebCalendarDB</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
    </properties>
  </persistence-unit>
</persistence>
```

Pro nás je důležitý tag <jta-data-source>, jeho obsahem musí být právě JNDI jméno databázového prostředku, s kterým má aplikace pracovat.

5.2.2 Konfigurace vzdáleného volání EJB

Vzdálené volání EJB komponent je standardně zakázané, nicméně my ho musíme povolit pro připojení aplikačního klienta. Nastavení vzdáleného volání najdeme v administrační konzoli v Configuration/ORB/IIOP listeners. Zde vybereme jeden z listenerů nebo si vytvoříme vlastní.

Důležité je specifikovat port, na kterém se bude poslouchat a zda se má využívat SSL (a jakým způsobem). Analogicky potom musíme nakonfigurovat i aplikační klient, který se bude připojovat k serveru. Po dokončení konfigurace musíme opět provést restart aplikačního serveru.

5.2.3 Deployment aplikace

Sestavení aplikace se provádí běžným způsobem pomocí nástroje ant. Sestavení se vždy provádí v EA (Enterprise Application) projektu – WebCalendar. Po jeho sestavení vznikne archiv .ear, který můžeme nahrát na aplikační server. Můžeme deployment provést buď z administrační konzole nebo z příkazové řádky. Při deploymentu aplikace může samozřejmě dojít k chybě – nejčastější příčinou je špatně nakonfigurovaná databáze (resp. JNDI jméno pro ni).

Deployment aplikace musíme provést před tím, než se pokusíme o konfiguraci aplikačního klienta.

5.3 Konfigurace aplikačního klienta

Pro spuštění aplikačního klienta je ideální mít na klientském počítači aplikační server Glassfish a použít ho pro spuštění naší aplikace. K tomu máme k dispozici skript appclient (popř. appclient.bat na Windows), který můžeme použít pro spuštění aplikace. Skript se postará o nastavení proměnných prostředí a hlavně inicializaci CLASSPATH, abychom mohli používat knihovny v aplikačním serveru (což jsou samozřejmě všechny knihovny J2EE frameworku – potřebujeme minimálně jejich API).

Další důležitou součástí je konfigurační soubor pro provoz aplikačního serveru. Obsahuje mimo jiné informace o tom, kde aplikační server běží, na jakém portu, zda se má používat SSL, popř. s jakým certifikátem. Obsah tohoto konfiguračního souboru musí korespondovat s příslušnými nastaveními v aplikačním serveru, jinak se připojení nezdaří.

Ukázka konfiguračního souboru:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE client-container SYSTEM
 "file:install-dir/lib/dtds/sun-application-client-
container_1_2.dtd">
<client-container>
          <target-server   name="localhost"   address="localhost"
port="36855" >

    <security>
      <ssl
        ssl2-enabled="false"
        ssl3-enabled="true"
        tls-enabled="true"
        tls-rollback-enabled="true"/>
      <cert-db path="ignored" password="ignored"/>
    </security>

    </target-server>
    <log-service level="WARNING"/>
</client-container>
```

Důležité jsou pro nás zejména tagy target-server, který specifikuje host a port serveru, na který se budeme připojovat a tak ssl, který určuje zda se má používat SSL zabezpečení připojení. Možnosti nastavení je samozřejmě celá řada, podrobný popis najdeme v dokumentaci Glassfish.

6 Závěr

Podařilo se nám vytvořit platformu pro vývoj aplikace typu organizér, zejména se nám podařilo vyřešit způsob ukládání dat, celkovou architekturu a hlavně způsob vzdáleného připojování jednotlivých uživatelů k této aplikaci.

Možnosti dalšího rozvoje jsou poměrně široké, např. by se dále mohly rozšířit podporované vlastnosti aplikace, jako např. spolupráce více uživatelů, což by z aplikace udělalo skutečné egroupware řešení. Další možností, která je dle našeho názoru neméně zajímavá, je vytvoření webového frontendu pro aplikaci. Výhodou aplikace je, že by byznys vrstva mohla zůstat naprosto stejná, pouze by se doprogramoval další frontend, který by ji využíval. Jako zajímavou technologii pro vývoj webového rozhraní se jeví Java Server Faces, což je standardní technologie v aplikačním serveru Glassfish pro RAD vývoj webových aplikací.

Vytvořeny byly též základní výukové materiály ve formě slajdů, které se dotýkají tématu platformy Java EE. Seznamují studenta se základními principy této platformy a měly by sloužit jako základ pro budoucí výuku programování aplikací na této platformě. Slajdy by se samozřejmě mohly dále rozšiřovat, Java EE obsahuje velké množství technologii, které jsme nevyužili (např. JMS, veškerá API pro XML, webové služby, bezpečnostní mechanismy atd.) a v rámci budoucího pokračování by rozhodně bylo možné vytvořit materiály i pro tyto technologie, či o ni rozšířit samotnou práci, která může sloužit jako dobrý zdroj a ukázka fungování těchto technologii v praxi.

Literatura

[01] Firebird SQL [online]. 2009, [cit 2009-05-20]. URL: www.firebirdsql.org

[02] JayBird [online]. 2009, [cit 2009-05-20]. URL: http://www.firebirdsql.org/index.php?op=devel&sub=jdbc&id=aboutjbird

[03] EclipseLink [online]. 2009, [cit 2009-05-20]. URL: http://www.eclipse.org/eclipselink/

[04] Hibernate [online]. 2009, [cit 2009-05-20]. URL: https://www.hibernate.org/

[05] Glassfish [online]. 2009, [cit 2009-05-20]. URL: https://glassfish.dev.java.net/

[06] JBoss [online]. 2009, [cit 2009-05-20]. URL: http://www.jboss.org

[07] IBM WebSphere [online]. 2009, [cit 2009-05-20]. URL: http://www.ibm.com

[08] Oracle WebLogic [online]. 2009, [cit 2009-05-20]. URL: http://www.oracle.com/technology/products/weblogic/index.html

[09] Oracle Containers for Java EE [online]. 2009, [cit 2009-05-20]. URL: http://www.oracle.com/technology/tech/java/oc4j/index.html

[10] SAP NetWeaver [online]. 2009, [cit 2009-05-20]. URL: http://www.sap.com/cz/platform/netweaver/index.epx

[11] Java SE technologies - Database [online]. 2009, [cit 2009-05-20]. URL: http://java.sun.com/javase/technologies/database/

[12] Wikipedia: ACID [online]. 2009, [cit 2009-05-20]. URL: http://en.wikipedia.org/wiki/ACID

[13] Jendrock, E.; Ball, J.; Carson, D.; aj.: The Java EE 5 Tutorial. Prentice Hall, 2006. ISBN 0321490290

[14] Java Persistence API [online]. 2009, [cit 2009-05-20]. URL: http://en.wikipedia.org/wiki/ACID

[15] Wikipedia: SyncML [online]. 2009, [cit 2009-05-20]. URL: http://en.wikipedia.org/wiki/Syncml

[16] Funambol [online]. 2009, [cit 2009-05-20]. URL: http://www.funambol.com

[17] Wikipedia: Microsoft Exchange Server [online]. 2009, [cit 2009-05-20]. URL: http://en.wikipedia.org/wiki/Microsoft_Exchange_Server

[19] Wikipedia: Multi tier architecture [online]. 2009, [cit 2009-05-20]. URL: http://en.wikipedia.org/wiki/Three-tier_(computing)

[20] TopLink JPA: How to configure primary key generation [online]. 2009, [cit 2009-05-20]. URL: http://www.oracle.com/technology/products/ias/toplink/jpa/howto/id-generation.html

[21] Borrie, H: Firebird 2.5 Release Notes [online]. 2009, [cit 2009-05-20]. URL: http://www.firebirdsql.org/rlsnotesh/rlsnotes25.html

[22] IBPhoenix: Classic or superserver? [online]. 2009, [cit 2009-05-20]. URL: http://www.firebirdsql.org/manual/qsg15-classic-or-super.html

[23] Wikipedia: HMAC [online]. 2009, [cit 2009-05-20]. URL: http://en.wikipedia.org/wiki/Hmac

[24] Welcome to NetBeans [online]. 2009, [cit 2009-05-20]. URL: http://www.netbeans.org/

[25] Swing Designer [online]. 2009, [cit 2009-05-20]. URL: http://www.instantiations.com/windowbuilder/swingdesigner/index.html

[26] Visual Swing for Eclipse [online]. 2009, [cit 2009-05-20]. URL: http://code.google.com/p/visualswing4eclipse/

Seznam příloh

Příloha 1. Popis databázového modelu
Příloha 2. DVD se zdrojovými soubory aplikace a slajdy obsahující následující adresářovou strukturu:

- thesis – PDF a ODF verze bakalářské práce
- source – zdrojové soubory aplikace
- slides – slajdy ve formátu ODF
- manual – uživatelský manuál ve formátu HTML

Příloha č. 1: Databázový model

Databáze pro náš projekt byla navržena více obecně než tak, jak ji ve skutečnosti využíváme – což by mělo sloužit pro budoucí rozšiřování aplikace a doplňování dalších funkcí.

USERS

Tabulka USERS obsahuje informace o zaregistrovaných uživatelích.
Obsahuje následující položky:

- user_id (PK)
  ID uživatele. Je generováno automaticky.
- real_name
  Skutečné jméno uživatele (většinou jméno přijmení – např. Pavel Palát)
- login
  Login informace uživatele
- user_pasword
  Hashované heslo. Konkrétní působ hashování závisí na aplikaci, databázová vrstva ho neřeší, pro ni se jedná pouze o binární data

CATEGORY

Tabulka CATEGORY obsahuje informace o vytvořených kategoriích pro úkoly a pro schůzky (události)

Položky:

- category_id (PK)
  ID kategorie. Generováno autmaticky.
- user_id (foreign key do USERS)
  ID uživatele který si tuto kategorii vytvořil
- category_name
  Uživatelské pojmenování kategorie.
- red
- green
- blue
  RGB hodnoty barvy, kterou uživatel zvolil pro danou kategorii v aplikaci

TODO

Obsahuje kalendářová data – seznam úkolů. Každý úkol má přiřazenou informaci o tom, kdy začíná, kdy končí a kdy má být připomenut uživateli. Úkol je vždy vytvořen nějakým konkrétním uživatelem

Položky:

- todo_id (PK)
  ID identifikující daný záznam (úkol)
- category_id (FK na CATEGORY)
  ID kategorie do které je úkol zařazen
- user_id (FK na USERS)
  ID uživatele vlastnící daný úkol
- start_date
  Začátek úkolu jako timestamp
- due_date
  Termín splnění úkolu.
- remind_date
  Datum a čas připomenutí úkolu uživateli
- state
  Stav úkolu. Jedná se o výčet následujících hodnot:
    - 0 – úkol nebyl zatím začat
    - 1 – úkol byl začat
    - 2 – úkol byl úspěšně dokončen
- name
  Jméno úkolu
- description
  Popis úkolu (zobrazí se v detailech v uživatelském rozhraní)

CALENDAR_ENTRY

Tato tabulka slouží pro uložení údajů o schůzkách (událostech) pro daného uživatele. Dá se říci že se jedná o nejsložitější tabulku v aplikaci, jelikož musí řešit problém opakování daných událostí.

Naše aplikace podporuje následující opakování:

- Žádné opakování
- Denní
- Týdenní
- Měsíční

U každé schůzky je stanoveno, kdy začíná, jaká je její délka v sekundách (to je nutné pro snazší počítání s opakujícími se událostmi) a datum kdy končí poslední opakování. Pokud je událost neopakující se, tak čas konce je roven začátku události + její délce. U opakujících se událostí je datum konce rovno konci posledního opakování.

U každé události je dále možno stanovit maximální počet opakování. Informace jsou zde redundantní z důvodu optimalizace vyčítání dat z této tabulky. Vyčítání je zde zvláště důležité optimalizovat z důvodu, že se jedná o nejčastější operaci a uživatel u ní bude očekávat rychlou odezvu.

Konkrétní výpočty které události se v daný den objevují jsou odpovědnosti byznys vrstvy.

Položky:

- calendar_entry_id (PK)
  Unikátní ID daného záznamu
- category_id
  ID kategorie události
- user_id
  ID uživatele vlastnícího daný záznam
- name
  Jméno události
- description
  Popis dané události pro uživatele
- start_date
  Začátek události (timestamp)
- event_duration
  Délka události v ms
- recurring_for
  Počet opakování
- recurring
  Výčet specifikující opakování:
  - 0 - Žádné opakování
  - 1 - Denní
  - 2 - Týdenní
  - 3 – Měsíční
- finish_date
  Datum posledního konce události

Na všech časových polích jsou z důvodu optimalizace výkonu vytvořeny indexy, což výrazně zlepšuje rychlost čtení, ale snižuje rychlost vkládání, což ale nicméně je pro naši aplikaci přijatelné, protože množství modifikací nad těmito fieldy je mnohem menší než počet čtení dat z této tabulky.

USER_TOKEN

Tabulka slouží pro ukládání o aktuálních sezeních uživatelů a jim přiděleným tokenům.

- token (PK)
  BigInt hodnota obsahující vygenerované ID tokenu
- user_id
  ID uživatele pro který tento token byl vygenerován
- last_used
  Datum posledního použití tokenu (používá se pro automatické odhlášení po uplynutí určitého timeoutu)

CONTACT

Tabulka obsahuje seznam kontaktů dané osoby v adresáři.

- contact_id (PK)
  Unikátní identifikátor daného kontaktu
- user_id (FK do USER)
  Uživatel vlastnící tento kontakt (kontakt je součástí jeho adresáře)
- name
  Jméno, pod kterým je uživatel uložen a zobrazován
- address
  Adresa kontaktu
- note
  Poznámka k němu
- email1
- email2
  Emailové adresy
- tel1
- tel2
- tel3
  Telefonní čísla
- fax

- icq1
- icq2
- jabberIm
- otherIm

  Kontakty na různé IM komunikační nástroje