# Aplikace „Sleduj svého poslance“, základní popis:

Systém se bude zaobírat monitorováním činnosti práce veřejných činitelů. Systém bude veřejně přístupný, pro běžné uživatele, a pokud bude mít uživatel práva administrátora, bude mít možnost do systému vkládat komentáře, informace a dokumenty. Nicméně návrh kompletního systému je na absolventskou práci příliš rozsáhlý, je tedy potřebné přistoupit k zjednodušení, tedy vytvořit jenom základní funkční kostru, která bude otevřená dalšímu rozvoji.

## A. Cíle aplikace

Považuji za důležité si stanovit na začátku cíle a mít je počas celého projektu na mysli.

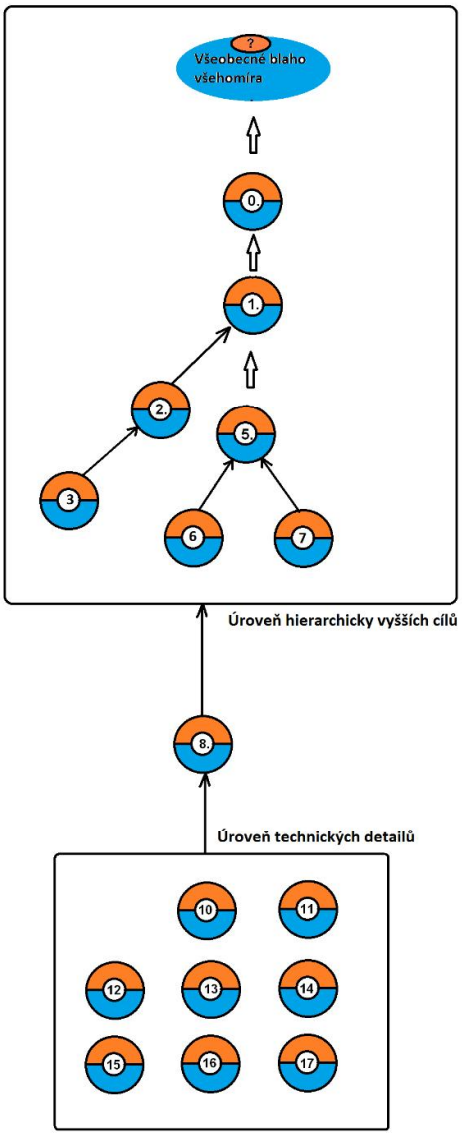
Cílů může být tolik, kolik je úhlů pohledů. Spíš než o cílech jednotlivě bude lépe, když budeme mluvit o ***systému cílů***, který má svou vnitřní strukturu a hierarchii. Následující očíslované body se náš *systém cílů* pokusí načrtnout, nicméně z povahy věci plyne, že tento nebude úplně vyčerpán.

**K ostrému dělení na požadavek / návrh řešení.** Při pozorném pohledu zjišťujeme, že neexistuje ostrá hranice mezi cílem (účelem, požadavkem) a prostředkem k dosažení cíle (návrhem řešení požadavku), ale že se spíš jedná o 2 různé aspekty té samé „věci“. Tedy jestli je nějaká věc cílem, nebo prostředkem, záleží jen na úhlu pohledu. Prostředky na dosáhnutí cílů vyšší úrovně jsou zároveň cílem (protože definují CO se má udělat), který dosáhneme zase jinými prostředky nižší hierarchické úrovně. Tedy již z podstaty věci ostré absolutní oddělení nemůže být realizováno. Tedy není možno od požadavku žádat, aby byl jenom požadavkem, protože je zároveň i řešením něčeho jiného. To samé platí i o řešení. Oddělení je možné jen pokud se týče relativního pohledu (tj. tenhle relativní cíl má takový-hle návrh řešení). Stejná dichotomie (*cíl / prostředek k jeho dosažení*) platí i pro dvojici Požadavek / Návrh řešení požadavku, protože to jsou jen jiné názvy toho samého.

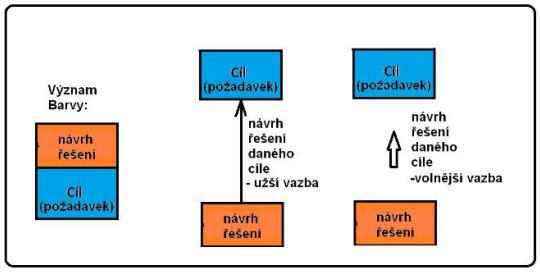
Graficky je ***systém cílů*** našeho sytému je znázorněn na obrázku č.1. Šipky na obrázku třeba chápat víceméně symbolicky, protože cíle se ve skutečnosti vzájemně překrývají a naopak, dané řešení může sloužit k dosažení mnoha cílů. Nicméně je možné si všimnout, že tyto cíle/ řešení je možné smysluplně uspořádat do určitých, řekněme sfér. Jelikož zaměření této práce je spíše praktické, bude lépe, když obrátíme pozornost na nižší sféry cílů / prostředků a to konkrétně týkající se implementace, nicméně pro úplnost, abychom si lépe uvědomili z čeho vycházíme, uvedeme i cíle hierarchicky vyšší.

Nicméně parametry se nevztahují k oběma zmíněným aspektům, např. ***priorita*** se vztahuje pouze k aspektu cíle. Bude-li tedy tato uvedena, bude se tedy vztahovat vždy jen k němu.

Co se týče formálního zápisu, hierarchicky vyšší cíle budu jenom vyjmenovávat, zatím co ty elementární – hierarchicky nejnižší, budu uvádět ve formě tabulky, jak se to běžně dělá.




**obr. 1. Systém cílů**




## Cíle-řešení:

(0). Aplikace podpoří demokratické principy v společnosti.

(1) A. Aplikace má zvýšit transparentnost konání vládních činitelů ve společnosti.

(2) B. Aplikace má být objektivní.(podporuje, (tj. je parciálním řešením) pro 1.). L2-vysoká (L1/L2 - vysvětlení viz níže).

(3) B.1. Objektivity chceme docílit tak, že aplikace bude umožňovat sbírání dat o konkrétních činech, resp. úkonech mocenského chování (tj. chování při hlasování) dané veřejně činné osoby tj. její výkonu veřejné služby (Naše řešení cíle č. 2).

(5) C. Rozšířenost. Aplikace má mít co největší dosah (tj. rozšířenost mezi lidmi). L2-vysoká.
- Systém je přístupný skrz běžný web. prohlížeč, ve formě zadané internetové stránky (např. www.kosvopo.sk).

- Systém bude běžet na serveru v režimu 24/7. Měl by být dostupný všem uživatelům dobré vůle, z jakéhokoli místa sítě.

(6) C.1. Aplikace bude uživatelsky příjemná. L2-vysoká.
Myšleno z hlediska uživatele i osvojovatele.
Z mého pozorování ve většině případů je celková nepřehlednost systému jakousi sumou mnohých drobných nepřehledností, které sice sami o sobě jdou poměrně lehce překonat, nicméně v masové kombinaci s dalšími vytvářejí onen prales, kterým se musí osvojovatel prosekávat. Je to analogie tření při pohybu tělesa o podložku, resp. přísloví „babka k babce, budú kapce“. Jakási průběžná nedbalost při odstraňování překážek pro budoucí osvojovatele (tj. Nemyslení na něj) tak vede k celkové nepřehlednosti systému. V takéto zbytečné nepřehlednosti (tj. Nepřehlednosti která nevyplýva ze složitosti samotného systému) vidím plýtvání duševními schopnostmi osvojovatele, protože jeho energie by měla být využita lépe, spíš na rozvoj systému, než na jeho osvojení. Zdá se mi, že když se k přehlednosti bude přistupovat systematicky, po malých krocích (kupř. přehlednost označení, názvů, etc.) může dojít snadněji k efektu „odtrhnutí laviny“ a člověk který systém studuje se do něj vtělí snáz a rychleji.

(7) C.2. Aplikace bude dostupná jako webová non-stop služba. (Řešení pro 5. a zároveň cíl pro body uvedené v části implementace). L2-vysoká.

(8) D. Realizace. Implementace. Aplikace bude schopna zabezpečit svůj vlastní chod (zde mluvíme o „fyziologických“ nevyhnutelnostech aplikace). Tento cíl je sice umělý, ale umožňuje odstínit (zkoncentrovat) všechny parciální cíle týkající se technických řešení od více abstraktních cílů (pozn. oto dělení je taky umělé). L2-vysoká.

## Úroveň technických detailů:

| č. požadavku: | 10 |
|---|---|
| název: | Ukládání historie. |
| uživatel: | dobrovolník, administrátor |
| popis: | Aplikace umožní uchovávat historii aktů veřejných činitelů a veřejných orgánů. |
| priorita: | L1-vysoká |

| č. požadavku: | 11 |
|---|---|
| název: | Ukládání změn v systému. |
| uživatel: | dobrovolník, administrátor |
| popis: | Aplikace umožní uchovávat historii aktů správců, tj. (uživatel dobrovolník a administrátor). Tedy ani zrušené entity se nebudou mazat, ale uchovávat. Stejně tak, jako i záznamy o každé změně. |
| priorita: | L1-nízká |

| č. požadavku: | 12 |
|---|---|
| název: | Vkládání dokumentů |
| uživatel: | dobrovolník, administrátor |
| popis: | Aplikace umožní k příslušným událostem vkládání pdf dokumentů a obrázků. |
| priorita: | L1-vysoká |

| číslo požadavku: | 13 |
|---|---|
| název: | Vkládání komentářů |
| uživatel: | dobrovolník, administrátor |
| popis: | Aplikace umožní dobrovolník-ovi vkládání komentářů k událostem. |
| priorita: | L1-vysoká |

| č. požadavku: | 14 |
|---|---|
| název: | Správu entit (objektů) |
| uživatel: | dobrovolník, administrátor |
| popis: | Aplikace umožní správu (vytváření/naplňování daty/mazání) entit (objektů), které jsou popsané v DB modelu. |

| č. požadavku: | 15 |
|---|---|
| název: | prohlížení profilu veřejných osob, etc. |
| uživatel: | Všichni |
| popis: | Systém bude generovat přehled údajů a historie působení jednotlivých veřejných činitelů / |

| | |
|---|---|
| | |
| priorita: | L1-vysoká |

| | |
|---|---|
| | veřejných orgánů / a ostatních entit<br>Tj. LOCATION, VOTE, a jiné... |
| priorita: | L1-vysoká |

| | |
|---|---|
| č. požadavku: | 16 |
| název: | 3-jí úroveň zobrazování |
| uživatel: | všichni |
| popis: | Aplikace bude podporovat trojí úrovně práv v systému. (tj. např. trojí zobrazení stránek, podle typu uživatele, atd..) |
| priorita: | L1-vysoká |

| | |
|---|---|
| č. požadavku: | 17 |
| název: | Přihlašovací procedura |
| uživatel: | dobrovolník, administrátor |
| popis: | Aplikace bude mít systém přístupu uživatelů (přihlášení, správa session, odhlášení) |
| priorita: | L1-vysoká |

## Priorita cílů (požadavků)

Taky priorita je relativní pojem. Co může rozhodovat o úspěchu z dlouhodobého hlediska, nemusí být podstatné při samotném spuštění a funkci aplikace. Majíce toto na mysli, napadá mně opět pyramida, tentokrát z oboru psychologie - *Maslowová pyramida lidských potřeb*. Potřeby, které jsou u základny pyramidy, jsou sice pro život naprosto nevyhnutelné, nicméně na směrování lidského života mají jenom nepatrný vliv. Bez potřeb z vrcholu pyramidy se sice dá lokálně přežít, nicméně jsou to právě ony, které určují směr civilizací, kultur a vůbec celého lidského žití. Jako příklad, bych uvedl: můžete rokovat s vládou třeba o záchraně světa, no když se vám „chce“ – musíte si odskočit, děj se co děj - má to vyšší prioritu. Nicméně i záchrana světa má svou váhu, asi vyšší než onen úkon. Proto tohle rozdělení.

Maslow rozděluje potřeby do vícero vrstev, pater. Prozatím (v rychlosti) bych si nedovolil navrhnout (pojmenovat) více pater, a proto volím jenom dvě základní.

Úroveň 1 = základní, bez kterých aplikace nebude fungovat. (L1)

Úroveň 2, - věci důležité z dlouhodobého, strategického hlediska, bez kterých však aplikace fungovat bude (tj. nač bude fungovat, když ji nikdo nebude používat?). (L2)

V obou úrovních pak budou stupně: Vysoká - zásadní, Střední - doplňující, Nízká - drobnosti.

## B. Uživatelé a lidský faktor

Systém budou užívat 3 typy uživatelů.

0. **administrátor**. Kromě práv dobrovolníka bude mít přístup k historii změn a možnost do systému přidávat / odstraňovat dobrovolníky. Vyžaduje přihlášení.

1. **dobrovolník.** Členové neziskových organizací, které se zabývají kontrolou činnosti veřejně činných osob (poslanců, politiků), aby v případě jejich nekalé činnosti upozornili širší veřejnost. Bude mít kromě práv občana, taky možnost vkládat a odstraňovat entity, dokumenty, události , komentáře, hodnocení. Vyžaduje přihlášení.

2. **občan.** Role je určena pro běžného návštěvníka stránek. Má privilegia jenom k prohlížení stránek (nemá přístup do všech). Systém tedy nebude podporovat veřejnou diskusi k tématům.

## C. Návrh řešení.

Úkol je možné řešit různými způsoby, jedním z cílů této práce je však vyhnout se „klasickým“ způsobům tvorby webových stránek, tj. za použití kombinace PHP, JavaScript a HTML a vyzkoušet některý z frameworků, které se v poslední době hojně rozmohly. Jejich hlavní výhodou je, že umožňují programátorům, zvyklým na vyvinutější programovací jazyky (Java) tvorbu webových aplikací, přičemž je odstiňují od základnějších technologií na kterých jsou ve skutečnosti postaveny. Programátor se tak nemusí rozptylovat obeznamováním se s detaily Javascriptu a HTML a zároveň je mu umožněno využívat silných stránek jemu známého jazyka, který tvoří vrstvu ve které jsou tyto základnější technologie obaleny. Tímto způsobem může poměrně snadno tvořit i stránky s komplikovanou funkcionalitou, co by bylo kupříkladu v prostším PHP problematické.

## Ukládání dat.

Standardním řešením je relační databáze založena na jazyku SQL, případně jeho dialektu. Vzhledem k tomu jak byla úloha zadaná, tj. nejsou kladeny speciální nároky jak na objem ukládaných / zpracovávaných dat a nejsou kladeny speciální nároky ani co se týče stupně bezpečnosti práce s daty. Nevidím smysl spekulovat nad jinými možnostmi (např. XML databáze), každé další řešení je za této konstelace zbytečná komplikace a proto se nad jinými typy ukládání dat nebudu ani zamýšlet. Data tedy budou ukládány ve formě databázových tabulek.

## Uchovávání vnitřní historie, tj. historie akcí apod.

V daném systému bude vhodné, aby se relevantní neaktuální záznamy nemazaly, ale byly potenciálně stále k dispozici. Na druhé straně nemohou být pro běžného uživatele viditelné. Tento problém budeme řešit zavedením pomocné proměnné „visible“, která bude v databázi uchovávat stav viditelnosti dané entity pro běžného uživatele.

## Spravovatelnost systému

S postupující komplikovaností softwarů dochází k otázce rozvržení projektu tak, aby bylo co nejpřehlednější, snadno udržovatelný a přitom co nejfunkčnější. Jádrem tohoto dění je izolace a separace podobných procesů, tj. funkcionalit. Pokud danou funkcionalitu v systému izoluju, mohu ji, jako celek vyměnit za jinou, bez toho, aby to zasáhlo další části programu. Takový kód je potom daleko přehlednější a jeho udržování je mnohem snadnější. V programátorské praxi dominuje izolační architektura (resp. návrhový model, což je častější označení) MVP (Model - View - Presenter).

Návrhový model MVP (Model-View-Presenter) vychází z modelu MVC (C=controller)

„Model“ v zjednodušení část Modelu by měla zahrnovat-popisovat základní vztahy vnitřní logiky programu, tj. vztahy mezi DB entitami, a tzv. bussiness logiku, tj. jak se má s vnitřními datami při té - které příležitosti zacházet, definovat mapu vnitřních stavů systému a přechodů mezi mini.

„Presenter“ – zahrnuje vrstvu, která se stará jak o ovládání změn dat v modelu, povětšinou na základě požadavků uživatele, tak i o přenos, resp. “zadrátování” změn hodnot dat v modelu tak, aby se projevili do změn ve „View“, který představuje zobrazovací i ovládací prvek, většinou s výstupem na monitor.

Ve všeobecnosti možno hovořit on MVP v případe „těžkých“ View, tj. kdy stav View závisí na stavu Modelu netriviálně, kde nelze jednoznačně rozlišit mezi vstupem a výstupem. Např. ukázkovým případem jsou Web stránky, např. tlačítko na web stránce může sloužit jak na výstup (objeví se, nebo zmizne) jako i vstup (stlačením dává uživatel pokyn). Naopak MVC je aplikovatelný v případě odlehčených“ View, kde jsou vstupy a výstupy striktně odděleny. Jako příklad bych uvedl fyzický knoflík jako vstup a diodu jako výstup – indikuje změnu stavu tak-řečeno napřímo.

Model v našem případě bude představovat kód řídící komunikaci s databází a business logiku systému.

Presenter se bude starat o přenos dat z Modelu do View, který představuje samotnou zobrazovací / ovládací část. Tento model se budu snažit zachovat i já.

### Přihlašování

Přihlašování musí splňovat několik všeobecných podmínek, které jsou na přihlašování kladeny. Jsou to bezpečnost při odesílání hesla z formuláře zabezpečenou linkou a bezpečnost při ukládání hesla do ďatabáze. Opět, jelikož ukládané data nejsou zvlášť choulostivé, budu vybírat jenom ze standardních řešení v rámci frameworku.

## Návrh databázového modelu

### Entity týkající se bussiness modelu:

PUBLIC_PERSON, PUBLIC_ROLE, TENURE, PUBLIC_BODY, LOCATION(OKRES, KRAJ), VOTE, VOTE_OF_ROLE, SUBJECT, THEME, VOTE_CLASSIFICATION, PERSON_CLASSIFICATION, DOCUMENT, NOTE

### Pomocné entity, resp. týkající se administrace:

USER, ROLE, USER_ROLE, CHANGE, HIERARCHY

#### 1. PUBLIC_PERSON (veřejně činná osoba)

Představuje fyzickou osobu, tj. člověka, který počas svého života může vykonávat vícero veřejných funkcí, čili rolí.

#### 2. PUBLIC_ROLE (veřejná funkce, resp. veřejně činná role)

Představuje konkrétní výkonní funkci působící ve veřejném životě. Tato je jednoznačně definována osobou, funkčním obdobím a veřejným orgánem (tj. PUBLIC_PERSON, TENURE a PUBLIC_BODY, od kterých „dědí“ identitu).

### 3. TENURE (její funkční období)

Funkční období ve vztahu ke PUBLIC_ROLE. Jedno funkční období může patřit i vícero funkcím. Předpokládám, že v praxi bude běžné že jedno funkční období bude příslouchat všem veřejných funkcí daného veřejného orgánu (tj. "od voleb do voleb"). Funkční období je jako samostatná entita jenom proto, aby jak zadání káže, bylo možné vyhledávat podle jednotlivých funkčních období. Jinak by zřejmě stačilo vložit do entity PUBLIC_ROLE povinný parametr *since* a nepovinný *till* (resp. pokud se vždy dopředu ví jaké bude mít daný mandát trvání, tak může být i povinný).

### 4. PUBLIC_BODY (veřejný orgán, ustanovizeň)

Představuje veřejný orgán, který rozhoduje o předmětech hlasování (SUBJECT).

### 5. LOCATION, DISTRICT, REGION

LOCATION je Místo, kde se veřejný orgán nachází. V konečné verzi si to představuji graficky: nejdříve bude mapa Slovenska se třemi aktivními částmi (kraji). Po zvolení některého z nich se uživatel dostane do přiblížení s aktivními okresy daného kraje, pak obcemi, resp. v některých případech městskými částmi. DISTRICT(okres) a REGION(kraj) představují uzemní zařazení kam daná obec patří. Sice platí, že většina okresů se odvozuje od okresního města(tj. stačila by jen entita LOCATION s odkazem na jinou - okresní město), nicméně všude ve světě tomu tak není, a podobně jako u krajů (na Slovensku se odvozují od krajských měst, v Čechách je tomu jinak) okres může být definován volněji. Proto je struktura LOCATION-DISTRICT-REGION vhodnější.

### 6. VOTE (hlasování orgánu o věci)

Představuje konkrétní jendo hlasování veřejného orgánu. Některé atributy VOTE, jako např. počet hlasů Za / Proti / etc. resp. celkový výsledek se můžou jevit jako nadbytečné, protože informaci v nich uloženou možno dopočítat z výsledků hlasování jednotlivých funkcí (a při tvorbě databáz je dobré se zdvojování informací vyhýbat, nakolik se osvědčilo skladování dané informace jen na jediném místě v DB). Na druhé straně se zvýší přehlednost a řešení případného rozporu bude lehce dohledatelné, takže se za tyto atributy přimlouvám. Atribut *internal_number* bude odkazovať na číslování, které používá daný orgán, taky kvůli přehlednosti.

### 7. VOTE_OF_ROLE (hlasování veřejně činné funkce)

Úkolem této entity bude uchovávat informace o hlasování dotyčné fyzické osoby, v rámci hlasování orgánu (VOTE) zprostředkovaně skrze funkci. Identitu dědí z entit PUBLIC_ROLE a VOTE.

### 8. SUBJECT (předmět hlasování)

Konkrétní věc, o které se hlasuje. Nepovinný je jeho vztah k témě. Zvolil jsem, že jej navrhuje, tj. předkládá na hlasování právě 1 osoba, která bude členem orgánu, který jej schvaluje.

### 9. THEME (kauza, resp. tématický okruh)

Širší tématický okruh, který se buďto řeší ve vícerých krocích hlasování - tj. kauza, nebo včeobecná témata například: územní rozvoj, vzdělávaní, atd.

### 10. VOTE_CLASSIFICATION (hodnocení hlasování)

Implementaci mechanizmu vyhodnocování této části můžeme odložit na pozdější stadium vývoje projektu, nicméně tak nějak předběžně s ní můžeme počítat již teď. Tato entita bude překládat do lidsky srozumitelného jazyka, o čem vlastně dané hlasování vypovídá (kupříkladu: pokud se hlasování bude týkat vykácání zalesněné části města za výstavy již pátého nákupního střediska, nebo vil prominentů je jeho veřejný význam zřejmé jiný než návrh podporující všeobecně potřebnou věc. Od této klasifikace se pak bude odvíjet i klasifikace veřejné osoby, podle toho jak se při hlasoování zachová.

Kritéria hodnocení musí být takové, aby v co nejmenší míře mohli být považované za subjektivní a nemělo by jich být mnoho. Mohli by se např. hodnotit na škále 0 – 5. Napadají mne např. *„veřejná prospěšnost“.*

Každé hodnocení musí být podložené odkazy na fakta(dokument, nebo poznámku), které k tomuto hodnocení vedly.

Kvůli zvýšení objektivity vyhodnocování bych navrhoval, aby do úvahy přicházeli výhradně fakta (tj. návrhy hlasování a hlasovaní) uložené v modelu. Tedy model bude zachovávat presumpci neviny každé veřejné osoby, která bude v něm figurovat a tato může být kompromitována výhradně pouze vlastním

chováním, které monitoruje náš systém. Vyhneme se tím částečně výtce, že na někoho přednostně "snášíme materiál".

Úplně ideální by bylo, kdyby tyto vyhodnocení dělal nějaký důmyslný a přitom transparentní (zveřejněný) algoritmus. V modelu předpokládám, že každá událost má nejvíce 1 hodnocení. Z těchto údajů by se jiným algoritmem mohlo / mělo v pravidelných intervalech (např. měsíčně) vypočítávat celkové hodnocení dané veřejné osoby (*PERSON_CLASSIFICATION)*.

### 11. PERSON_CLASSIFICATION (hodnocení veřejně činné osoby)

Představuje hodnocení veřejně činné osoby. U osoby se bude kromě „veřejné prospěšnosti“ vyhodnocovat parametr *„stabilita“* neboli *"soulad sám se sebou"* (tj. jestli dotyčný nehlasuje podle toho, jak se vyspí, nebo jak zafouká vítr).

### 12. DOCUMENT (dokument)

Představuje dokument. Je možné ho vztáhnout k jakékoliv entitě.

### 13. NOTE (poznámka)

Představuje textovou poznámku. Je možné ji vztáhnout k jakékoliv entitě.

## Pomocné entity

### 14. USER (uživatel)

Uživatel přihlášený do našeho systému. Předpokládám, že široká veřejnost se kvůli lepší přístupnosti nebude muset přihlašovat. Takže návrh systému uvažuje s tím, že kdo bude přihlášen, bude mít právo vkládat údaje a zasahovat do systému.

### 15. ROLE(role přihlášeného uživatele)

Role v systému(samotná), která vyžaduje přihlášení. Tj. Dobrovolník a administrátor.

### 16. USER_ROLE(role přihlášeného uživatele)

Role přiřazena uživateli na dané časové období.

### 17. CHANGE (změna)

Tato entita je jednou z možností jak evidovat redakční zásahy. V této by se mohli evidovat všechny vložení od daného uživatele. Další podrobnosti v kapitole *Implementační detaily*.

## Dodatek

Ve fyzickém modelu (PH) nahrazuji složené primární klíče samostatnými ID. V ER modelu ponechávám tyto, kvůli zvýšení přehlednosti vztahů mezi entitami.
**Rozdělení tabulek.** Tabulky resp. Entity je buďto týkají samotného navrhovaného bussiness systému, nebo jsou pomocné, resp. popisují administraci. Kvůli větší přehlednosti proto tento fakt zahrnuji i do názvu tabulek. Zvýší se tím přehlednost.

# D. Implementace, použité technologie

## D.1 Maven

Apache Maven je nástroj pro správu, řízení a automatizaci buildů aplikací. Ačkoliv je možné použít tento nástroj pro projekty psané v různých programovacích jazycích, podporován je převážně jazyk Java. Název maven pochází z jidiš a znamená „znalec“. Maven byl vytvořen jako nástroj pro zjednodušení buildů pro projekt Jakarta Turbine. Hlavním impulzem pro vznik byla snaha o standardizaci a znovupoužitelnost buildovacích skriptů, která v tehdy používaném nástroji Apache Ant nebyla plně podporována. [1]

## D.2 Reflexe

Reflexe je schopnost programovacího jazyka zjistit za běhu informace o určitém objektu. Obecně, poněvač není jen OOP, je to schopnost zjistit informace o programu a jeho syntaktické struktuře. V

objektově orientovaném programování je program rozdělen do tříd, kdy jednotlivá třída popisuje vnitřní strukturu objektu a jeho vnější rozhraní. Na základně tříd je možné tvořit jednotlivé objekty. Některé jazyky mají schopnost za běhu zjistit informace o daném programu. Tato schopnost se nazývá reflexe, s jejíž pomocí lze získat za běhu programu informace o typu objektu. V objektově orientovaném programování se dá říci, že vše je objekt, tak je tedy objektem i třída a jiné datové typy, o kterých lze zjistit požadované informace. [2]

Výhodou reflexe například je, že programátor nemusí dopředu vědět, jaké parametry / metody daná třída / objekt má, ale dovede si je získat za chodu programu a podle potřeby spustit, resp. Jinak využít. Dává mu tím do rukou značně silný nástroj pro zobecnění kódu, silou porovnatelný ukazatelům v jazyku C. (kupř. reflexi využívá mnoho moderních programátorských nástrojů, jako je třeba Spring)

## D.3 Spring

Jaká je filozofie Springu? Nejdůležitějším slovem v souvislosti se Springem je: kontext. Jádro Springu je postaveno na využití návrhového vzoru *Inversion of Control* a je označován jako IoC kontejner. Tento návrhový vzor funguje na principu přesunutí zodpovědnosti za vytvoření a provázání objektů z aplikace na framework. Objekty lze získat prostřednictvím *vkladání závislostí (dependency injection)*, což je speciální případ Inversion of Control. Dependency Injection řeší vlastní způsob vložení objektů. Základní tři způsoby vložení objektů jsou Setter Injection, Constructor Injection a Interface Injection. Objekty vytvořené kontejnerem jsou nazývány *Beans*. Objekty jsou frameworkem vytvořeny typicky na základě načtení konfiguračního souboru ve formátu XML, který obsahuje definice těchto Beans.[wiki]

Jinými slovy, Spring Framework je založen zhruba na následovné filozofii: Mezi objekty jsou různé vztahy (*dependencies,* tj. odkazy na jedné instance na instanci jinou). Pojďme tento model zkonstruovat tak, že nejdříve vytvoříme všechny potřebné objekty (bez vzájemných vztahů) a pak v druhém kroku vytvořme tyto vztahy. Nebo ještě lépe: vytvořme tyto vztahy až v okamžiku, když je budeme potřebovat. Vytváření vztahů se v Spring hantýrce říká *wiring*.

(Normálně je to tak, že jednotlivé instance, pokud obsahují odkazy na jiné, tak si je obvykle sami vytvoří, např. v rámci konstruktoru. Vnitřní instance je tak „uzavřená“ ve vnější instanci a není k ní zvenčí přístup, co může mít řadu nevýhod)

Spring Framework se nezabývá řešením již vyřešených problémů. Místo toho využívá prověřených a dobře fungujících existujících open-source nástrojů, které v sobě integruje. Tím se stává jejich použití často jednodušším.[wiki]

## D.4 Vaadin

Tenhle framework jsem si vybral, protože byl právě pro tento typ aplikací navržen. Konkrétně pro nasledovní jeho vlastnosti:

- Komplexní kompatibilita s ostatními webovými technologiemi. Jako např. HTML.
- Modularita. Tj. vlastnost umožňující přidávat a odebírat již hotové celky - moduly, s minimální přidanou námahou na jejich „zadrátování“. Tato vlastnost umožňuje lehkou rozšiřitelnost (extensibility)
- Podporuje MVP model.
- Mnoho nejčastějších úkonů a témat (přihlašování, registrace, propojení dat s grafickými prvky) je již vyřešeno do standardní podoby, není tedy třeba znovu objevit kolo..

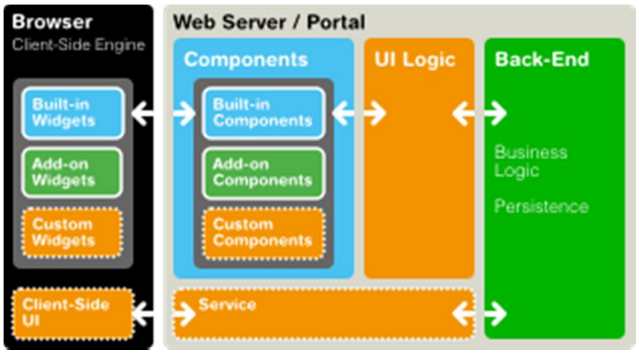
Jsem si však vědomi i jeho nevýhod:

- Ukrývání technologií, na kterých systém běží před zraky programátora. Tj. všechno je perfektní až do doby, když to přestane fungovat a hlavní výhoda se zvrátí v hlavní nevýhodu.
- Hotová řešení se někdy těžko upravují do podoby kterou zrovna vyžaduje situace.

Co je to Vaadin? Vaadin je „Java web developement Framework“ určený ke vytváření a udržování komplexních webovských aplikací. Podporuje 2 rozdílné programovací přístupy: server-side a klient-side. Nejsilnější stránkou Vaadinu je však právě tvorba aplikací pro serverovou stranu. Kde programátor prakticky využívá stejné nástroje jako při tvorbě jiných desk-topových aplikací. Nemusí se tedy vůbec seznamovat s detaily překladu do JavaScriptu a html, od kterých je úplně odstíněn.

V minulosti byla jistá propast mezi programátory „klasickými“ a programátory web. Stránek. Pokud klasický programátor chtěl udělat webovskou aplikaci, musel opustit jemu známý programovací jazyk a naučit se něco pro web charakteristických technologií (PHP, JavaScript a pod.). Jeho produktivita tedy šla dolů a nemohl se zaměřit plně jen na logiku aplikace. Vaadin se pokouší tuto propast překlenout a umožnit tvorbu web aplikací i programátorům používajícím jazyk Java.

Vaadin se tak stará o obsluhu uživatelského rozhraní a AJAX-ovou komunikaci mezi browserem a serverem. Viz. Obrázek 1.1.




Obr. 1.1

Tento obrázek ilustruje základní architekturu server-side aplikací. Architektura pozůstává ze server-side frameworku a klient-side engine, která běží na browseru, poskytující uživatelské rozhraní a doručující uživatelské akce na server. Uživatelské rozhraní aplikace běží jako Java Servlet Session na Java aplikačním serveru a klient-side engine jako JavaScript.

Protože klient-side engine je vykonávána jako JavaScript v browseru, nejsou pro použití Vaadinu nutné žádné dodatečné plug-iny do browseru. Toto poskytuje výhodu oproti jiným Framework-ům využívajících plug-iny (Flash, Java Applets, etc.. ).

Vaadin je závislý od podpory pro GWT (Google Web Toolkit), která je však běžná pro široké spektrum browserů. Takže developer se nemusí vůbec starat o podporu ze strany toho-kterého prohlížeče.

Při tvorbě aplikací pro klientskou stranu vaadin využívá GWT, ktorý poskytuje compiler z Javy do JavaScriptu, který běží na prohlížeči. Tedy tak, nebo tak developer přichází do kontaktu jedině s Javou.

Vaadin taky podporuje taky jasné oddělení mezi strukturou uživatelského rozhraní a jeho vzhledem, a umožnuje je vyvíjet separátně, nezávisle na sobě. Vaadin řídí vzhled stránek kompletně skrz tzv. *témata* které využívají CSS dokumenty, případně šablony HTML stránek.

Základní třídou ze které se ve Vaadinu vychází je třída, která dědí z třídy *com.vaadin.ui.UI*. UI je částí webovské stránky ve které Vaadinovská aplikace běží. UI je sdružená se uživatelovým *session*, které je vytvořeno pro každého uživatele pracujícího s aplikací.

Základní metodou této třídy je *init*(), která se spouští automaticky při prvním vstupu do aplikace.

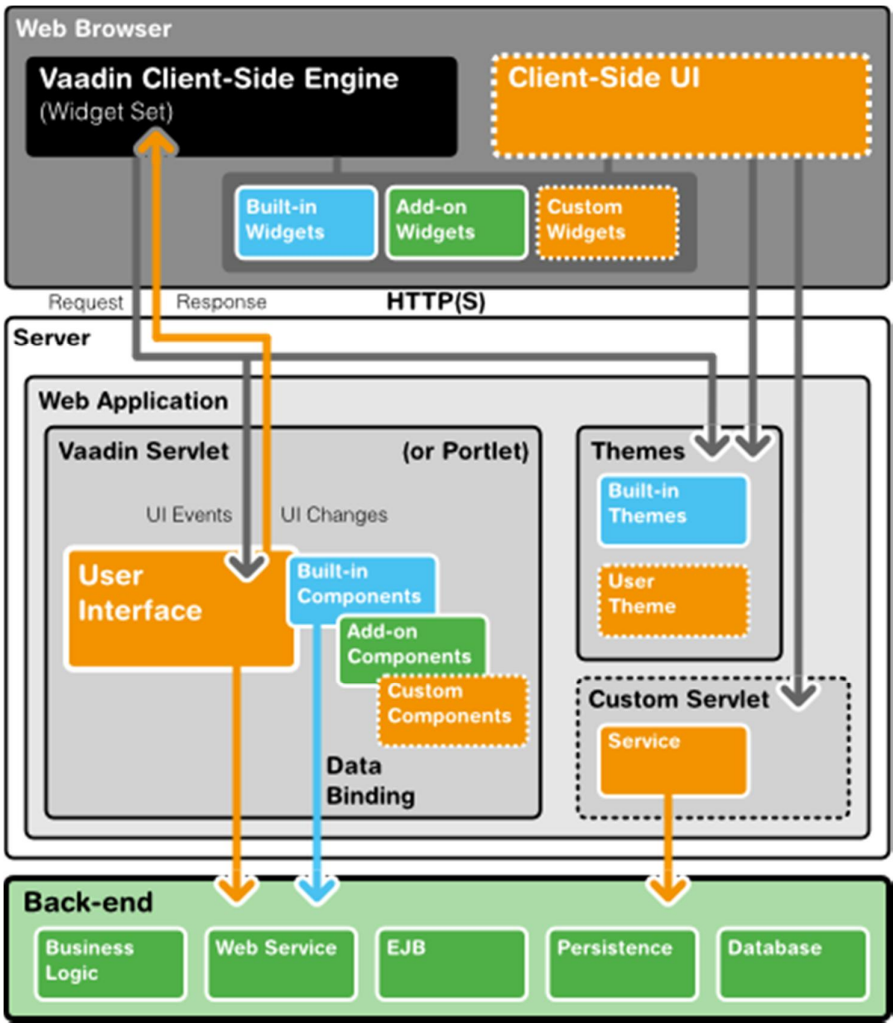
## Trochu historie

Vaadin je víceméně produktem plynulého vývoje myšlenky vývojového prostředí pro tvorbu webových aplikací. Tento vývoj řekněme, že začal v roku 2000 produktem skupiny IT Mill. Tehdy měl jeho předchůdce název Millstone library. Tato měle široké využití v rámci společnosti IT Mill, na tvorbu stránek pro své klienty. Další generace, IT Mill Toolkit Release 4 v roce 2006 zavedla kompletně nový

systém (engine) založený na AJAX-u. Tento umožňoval tvorbu aplikací bez starostí o komunikaci mezi klientem a serverem. Dalším milníkem byla verze IT Mill Toolkit Release 5, která pokročila ještě hlouběji do AJAX-u a klientská část bylo kompletně přepsána pomocí GWT, Google Web Toolkit. Od tohot momentu bylo možné použít Javu jak na serverové, tak i klientské straně. K šestému vydání IT Mill Toolkit-u nedošlo, nicméně v roce 2009 bylo vydáno pod novým jménem jako Vaadin 6. Zde nastává exploze uživatelů frameworku. Vydání Vaadin 7 v roce 2012 bylo zatím posledním krokem. Ve Vaadinu 7 došlo k úplnému zahrnutí GWT.

## Architektura Vaadinu

Se zaměřením aplikací na serverově straně. Nasledující obrázek ukazuje základní vztahy mezi aktéry komunikace. Vaadin Framework pozůstává ze API serverové strany, API klientské části a množství komponent uživatelského rozhraní na obou stranách. Témat, ovládajících vzhled a data modelu, implementaci logiky systému a vázání dat přímo do komponent systému. Klientská část zase zahrnuje Vaadin kompilátor, umožňující kompilaci Javy do JavaScriptu.



Serverová část aplikace běží jako servlet na Java web serveru, obsluhujíc http žádosti. Zvyčejně k tomuto účelu bývá využit VaadinServlet. Server přijímá požadavky od klienta a interpretuje je jako události dané klientské session. Události jsou asociovány s komponenty uživatelského rozhraní a doručovány k listenerům těchto událostí. Pokud klient způsobí změnu na serverové části, servlet je předá internetovému prohlížeči, který vytvoří odezvu pro uživatele aplikace. Tuto odezvu zachytí engine na klientské části a použije je na provedení změn internetové stránku, kterou uživatel zrovna prohlíží.

### Technologické pozadí, HTML a Javascript, CSS

Téměř celý web je postaven na technologiích HTML, který definuje strukturu stránky. Definuje jak grafickou i hierarchickou strukturu textu navíc umožňuje vkládání odkazů a obrázků. Vaadin používá XHTML, který je syntakticky přísnější. Používá verzi HTML 5. JavaScript na druhé straně je programovací jazyk, který pracuje v součinnosti s HTML stránkami a je možné ho do nich implementovat. JavaScript může manipulovat s HTML stránkou skrz DOM (Document Object Model). Klient-side engine a klient.side widgets jsou zkompilovány právě do JavaScriptu, pomocí Vaadin Klient Compiler-u.

Vaadin zhusta skrývá použití HTML, dovolujíc se programátorovi koncentrovat na logiku stránek. Při aplikacích pro serverovou část UI je vyvíjeno pomocí UI komponent a přeloženo pomocí client-side engine do podoby HTML stránky.

Z webových technologií je převzeto také tvorba stylu stránek. Používá se k tomu všeobecně rozšířený jazyk CSS (cascade style sheet). Technologie Sass(syntactically awesome stylesheets), také využívaná ve Vaadinu je rozšířením CSS. Umožňuje použití proměnných, v-hnízdění a mnoho dalších syntaktických črt, které dělají použití CSS přehlednější. Vaadin již má přichystané základní docela dizajnově propracované témata, kterých výhodou kromě toho že uživatel nemusí začínat tvorbu vzhledu stránek „na zelené louce“, ale např změnou, či rozvíjením jejich motivů je, že umožňují graficky identifikovat technologii. Z tohoto důvodu mnoho uživatelů základní témata nijak nemění.

### AJAX

Znamená Asynchronous JAvaScript and XML je technologie pro tvorbu webových aplikací podobným desktopovým aplikacím. Klasický přístup umožňuje načítat HTML stránku jen jako celek. Podstata této technologie spočívá v odesílání klientových požadavků na server asynchronně, tj. bez čekání na odezvu (tj. stránka běží dál, jako by se nic nedělo). Po přijetí odezvy ze serveru se příslušně upraví buďto celá stránka, nebo jen její část. Nedojde tedy ke „zmrznutí“ stánky, zatím co čeká na výstup ze severu. Asynchronní žádost v AJAXu umožňuje třída *XHTMlHttpRequest* v JavaScriptu.

## Psaní aplikací pro serverovou stranu

Aplikace serverové strany běží jako Java Servlet ve servlet kontejneru. Nicméně Java Servlet API je skryto za frameworkem. Uživatelské rozhraní je implementováno jako třída UI, která musí být vždy vytvořená.

## Základní prvky aplikací UI

Reprezentuje HTML fragment, ve kterém aplikace běží ve webovské stránce. Běžný postup je, že hlavní třída aplikace dědí z UI.java. Do této třídy se pak vkládá obsah stránky. UI je původně zobrazovací pole spojené s uživatelským session dané aplikace. Běžně když uživatel otevře novou stránku s URL daného UI, vytvoří se automaticky nová instance třídy UI a asociovaný objekt „Page“. Toto všechno je asociováno se session.

## Page

Je objekt asociovaný se UI, reprezentuje webovou stránku jako i okno prohlížeče, kde běží UI. Může být přístupné skrz `Page.getCurrent()`, nebo pomocí `UI.getCurrent().getPage()`.

## Vaadin Session

Reprezentuje uživatelský session. Začíná okamžikem, kdy prvně otevře UI, nebo spustí Vaadinovskou aplikaci. Končí buďto ukončením aplikace, nebo vypršením času session. Bude využito kupř. na rozeznání běžného uživatele od administrátora.

## Navigace

Vaadinovské aplikace nemají navigaci jaká je běžná u normálních stránek. Tyto běžně běží na jenom 1 stránce podobně jako všechny aplikace založeny na Ajaxu. Běžné je, že aplikace mají vícero View, mezi kterými uživatel přepíná. Tuto službu dělá ve Vaadinu třída `Navigator`. Views spravované pomocí něho automatický získávají specifické URI, které může být použito na přímou navigaci, skrz příkazový řádek v browseru.

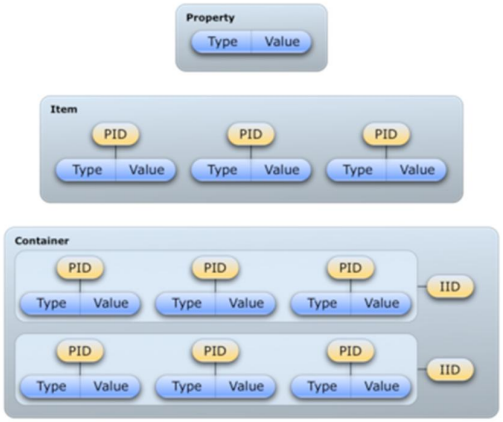
## Komponenty uživatelského rozhraní

Uživatelské rozhraní pozůstává z komponent, které jsou součástí aplikace. Jsou ukládány v hierarchickém uspořádání do tzv. *Layout* komponent, a hlavním, kořenovým layoutem, který je implicitně zastoupen v třídě UI. Nejdůležitější „koncové“ komponenty Vaadinu jsou co do typu prakticky totožné s komponenty knihovny AWT. Button, Table, TexField, ComboBox, Window, etc… Uživatel – programátor si může na základě existujících komponent sám vytvářet nové komponenty

## Vázání komponent a dat(Data *binding*)

Jednou z filosofií ve Vaadinu je odstraňovat prostředníky, co to jde. To se projevuje i v široké podpoře vázání dat s komponenty. Z pohledu MVP je to zkratka, nicméně se s MVP úplně nevylučuje (ale posouvá samotný model směrem k modelu MVC). Filozofie je ta, že změna dat, tak jak je vidí na obrazovce uživatel vede k přímé změně dat v modelu. Jsou tři vnořené úrovně hierarchie v data modelu Vaadinu. *Property, item* a *container*. Při použití analogie s datovým modelem tyto položky zodpovídají buňce, řádku resp. celé databázové tabulce.

*Data binding* představuje způsob, jak efektivně propojit View s Modelem. Vlastnost fieldu je přímo propojená s databází, a tak změna v daném poli se přímo projeví do změny hodnoty v databázi, nebo proměnné, která je na dané pole navázaná. Tento koncept, stojí na pojmech *Container*, *Property a item* kde container představuje pole hodnot, které se můžou vázat ke komplikovanější komponentě, např. ComboBoxu. Property je naopak jednoduchou hodnotou, která se váže k danému políčku. Tento koncept bude hodně využívaný i praktické části této absolventské práce. Vztah mezi *Item, property* a *containerem* je znázorněn na dalším obrázku.




## Zapojení aplikace („deploying“)

Vaadinovské aplikace jsou zapojované jako Java webovské aplikace, které můžou obsahovat množství servletů a statické zdroje, jako např. obrázky a HTML soubory. Takováto webovská aplikace je běžně zabalená do WAR souboru (s koncovkou `.war`). je to podtyp balíčku typu JAR (Java ARchive) a podobně jako obyčejný Jar soubor je in War ZIP-komprimovaný, se specifickou strukturou obsahu.

V hantýrce Java – Servletů *webová aplikace* značí kolekce *servletů* a *portletů,* JSP a statických HTML souborů a různých jiných zdrojových souborů, které tvoří aplikaci.

# E. Technické detaily řešení

## E.0 Řešení přístupu do databáze

S přístupem do DB je možné se vypořádat všelijak. Dostupné je množství řešení, která můžeme rozdělit do následujících kategorií.

A.) Níže-úrovňové, tedy ty, které navazují s databázovým serverem spojení, a jednotlivé operace převodu mezi světem javovských instancí a DB se komunikují v podobě SQL dotazů, které konstruuje uživatel.

B.) Výše-úrovňové Tyto frameworky odstiňují uživatele (programátora) od detailů komunikace s DB. Některé i od samotné tvorby SQL dotazů. Mně osobně jsou sympatičtější technologie typu A. Jsme totiž názoru, že jednoduchost znamená minimálními prostředky popsat realitu. Pokud je však sama realita komplikovaná, snaha zjednodušit její popis nutně vede k zahalování, odstiňování vnitřních funkčních částí. Výhodou je zjednodušené používání pokud vše funguje v pořádku. V okamžiku když však systém někde uvízne, dolehne na uživatele všechna tíha, o kterou si práci ulehčil. Narazí totiž na nepřehledné předivo nejasně viditelných vztahů schovaných pod kapotou. Požadavky na produktivitu však neumožňují příliš hluboko zacházet do přístupu typu A, a tedy dá se říci, že uživatel je nepřímo nucen volit přístupy typu B. Ideálním výchozím bodem pro přístup typu B je dostatečně dlouhá práce přístupem typu A, kdy uživatele – programátora nakonec unaví donekonečna vykonávat rutinné úkony a práci si nakonec zjednoduší. Touto cestou prakticky tyto frameworky vznikají. Proti tomuto tak řečeno „přirozenému“ vývoji není prakticky závažných námitek. Problém nastává v okamžiku, když do „zjednodušeného“ systému má nastoupit nezkušený uživatel. Ten, obrazně řečeno platí pomyslný dluh za uživatele, co daný systém sestrojil. Nastává tak situace, když pro stávajícího obyvatele světa IT technologií je život čím dále, tím jednodušší, zatímco pro nově začínající programátory je vstup do tohoto světa čím – dále složitější. Situace není nepodobná světu byznysu, kdy pro majetné společnosti je čím dál jednodušší dále bohatnout, zatím co pro začínající podnikatele je situace se uchytit čím dále namáhavější. Tato monopolizace vlastnictví IT poznání – podobně jako v ostatních odvětvích implikuje 2 základní přístupy, jak se s ní vyrovnat. 1. Přístup vede k „instantnímu přizpůsobení“ tj. jakémusi zrychlenému ztotožnění se modelem. Cenou je povrchnější znalost systému, která se však projeví až později v krizových situacích. Ty pochopitelně „hasí“ zkušení matadoři, upevňujíc svoje monopolní postavení. Druhý přístup spočívá ve snaze o hlubší pochopení systému, což je však časově náročné a stejně rozdíl oproti matadorům se stírá jen částečně. Mají prostě již moc velký náskok. Naštěstí, na rozdíl od byznys-přirovnání, vědomosti není možné v plné míře zdědit, ale jen v omezené míře předávat

dál, čím je zabezpečeno jaké – také znovunastolení rovnosti příležitostí v případě matadorova úmrtí. To je výhoda těch, co jsou v čele peletonu.

Jelikož není na škodu, když absolventská slouží i jako edukační prostředek (mám na mysli pro toho, kdo ji píše) v projektu jsem zvolil stezku A.

### Filozofie connections a transakčních operací.

Ve vztahu k databázi se v systému vyskytují buďto 1. neinvazivní dotazy (tj. jenom selecty)), nebo dotazy invazivní (měnící obsah databáze). Invazivní dotazy bych ještě rozdělil na dotazy jednoduché (jenom 1 operace) a transakční (2 a více operací, které se musí provést buďto všechny, nebo žádná).

Těnto typům operací bude přizpůsobena i správa databázových připojení.

A. Pro všechny neinvazivní operace bude jedno společné connection a metody k jeho přístupu budou synchronizovány.
B. Pro všechny invazivní jednoduché dotazy bude získáno separátní connection(z connection pool-u), které po provedení operace provede commit a bude opět uvolněno.
C. Pro transakční operace bude získáno separátní connection(z connection pool-u), které po provedení všech operací provede commit a bude opět uvolněno.

## E.1 Odstaňování / deaktivace entit

Jelikož entity nejsou na sobě nezávislé, ale mají určité vazby, bude nevyhnutné tyto vazby zohlednit i při jejich odstraňování, tj. Deaktivaci, protože pochopitelně nemá smysl držet v systému entitu, která odkazuje na jinou, již zrušenou. Využijeme k tomu stejný algoritmus jako v případě výše uvedeného filtrování.

Na obrázku č.XZY je uveden strom hierarchických vazeb entit vyskytujících se v našem systému.

Jedná se o stromový graf, kde kořeny jsou entity úplně vespod a listy úplně nahoru. V programu mám tento vztah nazvaný boss – slave.

Jeho jednotlivé vrstvy znamenají, čím nížeji je entita v hierarchii, tím má základnější postavení. Tj. Entita z 2. vrstvy nemůže existovat bez entity z vrstvy první na kterou ukazuje. V komplexnějším systému však narážíme na otázku: když odstraníme entitu A, které všechny entity by měli zmizet?

Algoritmus jejich vyhledávání při deaktivaci bude následovní:

1. Vyjdi z entity, kterou cheš odstranit.
2. Najdi všechny entity (co do počtu i typu) které na tuto entitu ukazují. Poznamenej si je.
3. Pro každou entitu z bodu 2. udělej postup popsaný v tom samým bodě (2).
4. Proces 3. opakuj dokud neskončíš na listové entitě (nikdo již na ni neukazuje).
5. Deaktivována musí být taky množina entit získaná sjednocením všech zaznamenaných entit z bodu 2.

Pozn. V některých případech se může stát, že po deaktivaci hlavní entity jsou podřízené entity z našeho systému nepřístupné (např. dokumenty dané entity). a tudíž jejich deaktivace se může jevit jako zbytečná akce, tj. porušení principu nezbytnosti akcí. Při podrobnějším pohledu zjistíme, že není tomu tak a ve skutečnosti méně energie mineme, když bezpečně deaktivujeme vše, co deaktivované býti má. Je to z toho důvodu, že one zákon nezbytnosti akcií, vychází z hlubšího zákonu "šetření energie", který je mu nadřazený. Pokud nezdeaktivujeme entity, který by stejně systém neviděl, musíme vydávat průběžně energii na to, aby se dalším vývojem aplikace nestali tyto jiným spůsobem přístupné. Tj. stále musíme zbytečně čelit této "hrozbě".

2. Hlavní výdej energie spočívá v tom, že musíme pracně rozlišovat, který strom entit deaktivovat celý a který jen částečně (protože podentity nejsou viditelné). Naopak, pokud deaktivujeme celý strom, tak jak je potřeba, administrátor při pohledu do databázy získá větší přehled o tom, co je v jakém stavu. Tj. zvýší se transparentnost, celého systému.

Vazby mezi entitami jsou dané samotnou definicí v databázy (cizí klíče). Pro urychlení však bude dobré si vtvořit strukuru, která tyto vztahy jednoznačně popisuje. Zvolil jsem formu další databázové tabulky A_HIERARCHY. Výhoda toho způsobu je v tom, že daný vztah deklarujeme. Tím pádem získáváme větší kontrolu nad celkovým chováním filtru (pokud chceme z jakéhokoli důvodu některý vztah ignorovat, prostě ho v tabulce neuvedeme).

## E.2 filtrování

Při řešení problematiky filtrování jsem narazil na 2 typy.

0. podle existenční závislosti, tj. Řekněme „podle potenciálního stavu“
1. podle (řekněme) ‘aktuálního stavu‘. – vysvětlím.

### Filtrování podle existenční závislosti

Při postupném zaplňování systému je třeba počítat s nepřehledným množstvím položek v jednotlivých tabulkách, nebo jednoduše potřebujeme z celého souboru entit vybrat jenom ty, které se hodí do vybraného pohledu. K tomuto účelu je nezbytné zavést filtrování. Jeho algoritmus bude založen na podobném principu jako odstraňování dokumentů z předchozí kapitoly, s tím rozdílem, že vyhledávání nepůjde od vybrané ovlivňující entity až na úroveň listů ale jen do takové hlobky, než zarazí na ovlivněnou třídu a zacházet se bude jenom se získanou množivou entit ovlivněné třídy.

Výhoda těchto postupů je v tom, že filtrování je naprosto univerzální, tj. Při jakékoli kombinaci: ovlivňující vs. ovlivněná třída. To platí i pro odstraňováni entit.

Při vkladání nových entit nás z logiky věci zajímá filtrování prvního druhu, protože když vkládáme novou entitu, zajímá nás jaké jsou všechny možnosti parametrů, které ji mohou přislouchat.

Naopak při pasivním prohlížení nás zajímají jenom parametry, které byly zvolené a zrovna v systému jsou.

### Filtrování podle aktuálního stavu

Pokud si zvolíme kupř. Veřejný orgán, který nás zajímá. Tak sice tématický okruh hlasování (školstvo, životní prostředí, etc.) je sice od tohto nezávislý, nicméně zajímají nás jenom ty na které odkazují hlasování uskutečněné v daném veřejném orgánu. Tento typ filtrování budu využívat při tvorbě jednotlivých pohledů – View.

## E.3 Ukládání změn

Při každé změně systému se tato zapíše do databáze do specielní tabulky na to určené (tabulka A_Change). Do této tabulky se uloží datum změny (timestamp), id osoby, která změnu provedla, název tabulky, kde změna proběhla, id řádku v tabulce, kde změna proběhla, stará hodnota, nová hodnota.

Jsou dvě možnosti jak změny ukládat a to na straně Javy, nebo na straně Databáze, tj. V rámci databázového stroje. Výhodou ukládání změn na straně databáze, je určitě rychlost.

O nevýhodách tohto spůsobu bych chtěl trochu pojednat.

Pokud existují 2 nezávislé systémy, které zasahují společnou oblast, ze zkušeností platí, že je dobré pokud je jeden systém jakoby hlavní a druhý jakoby podřízený. Tedy něco, že na jednom smetišti by měl být pánem jenom jeden kohout. Pokud tomu tak není, musí být vyřešena jejich synchronizace, protože jenom tak můžeme předejít zbytečným kolizím. Z praxe vychází synchronizace téměř vždy

nákladnější na prostředky v porovnáním s prostým vymezením sfér vlivu. Vymězení autonomních oblastí s pouze jedním „pánem“ bývá jednodušší, přehlednější a tímpádem více odolné vůči chybám.

Většina logiky se však řeší na straně Javy. Pokud by se část logiky přesunula na stranu DB, nutně by vznikli „kolizní plochy“. Nemuselo by tak tomu být, kdyby kupř. bylo žádoucí do DB tabulky zapsat úplně jakoukoli změnu. Jak však ukážu dále, některé změny zapsat do tabulky změn není vhodné a tudíž je situaci jednodušší řešit na straně Javy i za cenu zpomalení aplikace.

Taky by mohla vzejít námitka, že je zbytečné v A_CHANGE ukládat novou hodnotu, když je přece tato uložena v samotné (jiné) DB tabulce. To je sice pravda, ale uložení i této hodnoty celkově zjednoduší rekonstrukci předchozího stavu a taky může představovat formu zálohy dat, takže získá se tím robustnější a v neposlední řadě přehlednější řešení.

Systém bude taky nastaven tak, aby změna objemých dat (tj. dokumenty) nemohla probíhat (aby se zbytečne nezatěžovala tato tabulka). To je týká u nás jedině tabulky T_DOCUMENT ,tj. Pokaždé, když bude do systému vložen dokument, bude mu přiděleno nové id, a změna z hlediska uživatele, bude představovat přidání nového řádku v DB.

### Univerzální formát.

Jelikož v tabulkách jsou hodnoty uloženy v nejrůznějších formátech (VARCHAR, DATE, TEXT, BIT, etc..) na to, aby mohli být ukládány v jednom stloupci, je potřebné najít univerzální formát na který bude možno transformovat jakoukoli hodnotu a taky co nejjednoduššeji provést zpětnou transformaci.

Nejuniverzálnějším fromátem se jeví proud bytů, tj. Kupříkladu BLOB. Nicméně, pokud jsme ze systému vyloučili změny dat, které tento formát vysloveně vyžadují (tj. dokumnetů). Druhým univerzálním formátem je String, tj. VARCHAR, ze kterého jde snadněji převod zpátky na potřební formát. Při použití typu BLOB by stejně jeden mezikrokem musel být pravděpodobně typ *String.*

## Ukládání dokumentů

Dokumenty je možno ukládat v databáze jako typ BLOB, nebo jako odkazy k adresárové struktúře na serveru. Výhodou dokumentů uložených v adresářové struktuře je to, že jsou dostupné i jiným způsobem, a též, že odlehčují databázový provoz. Naopak ukládáni dokumentů v databáze se mi její jako elegantnější přenostitelnější řešení.

## Návrat do bodu v minulosti

Díky struktře ukládání změn bude návrat do minuloasti(stavM) jednoduchou iterací skrz tabulku A_change od současného dne(stav S) až do okamihu ke kterému se chceme vrátit, přičemž v každím

iteračním kroku provedeme reverzní krok změny. (tj. pokud byla stará hodnota v tabulce X 5 a nová 4, změní se hodnota v tabulce X na 5.) Tímto způsobem se dostaneme postupně až do stavu, který panoval v době, kterou jsme si zvolili. Model umožní taky deaktivovat i akce vybraných jednotlivých uživatelů, ale za tímto účelem bude potřeba zabezpečit současnou deaktivaci takových změn jiných uživatelů, které jsou od na změnách našeho uživatele závislé.

Důležité je tyto změny provést mimo historii uchovávací systém, jinak se bude zbytečně jako změna počítat i návrat do minulosti – docházelo by k nežádoucímu „zrcadlovému“ množení údajů v této tabulce. To je jeden z důvodů proč jsem nezvolil sledování změn skrz triggery na databázové vrstvě. Tento stav bude potřeba reflektovat i metod ukládání do databáze, tj. Něco ve smyslu: „do simple step only“ a „do complex step“.

Všechny tyto problémy vyplývají z toho, že náš navrhovaný systém ukládaní změn umožňuje jen jednu „větvu“ historie. Pokud tedy ve stavu do kterého jsme se vrátili, vykonáme další změny (tj. Stav D) a opět se budeme chtít vrátit do „budoucnosti“(stav S), promítnou se nám tyto změny také. Toto „prolínání“ větví se mi nejeví jako dobrý nápad (s největší pravděpodobností by musel systém značně zkomplikovat, protože by museli být řešeny kolize resp. Existenční závislosti mezi tím co bylo přidáno a „budoucí“ větví). Pokud tedy nechceme systém rozšířit o možnost pracovat ve větvích (analogie s verzovacími systémy např. GIT), musíme při každém vrácení se do minulosti krok ze kterého jsme se vrátili v tabulce CHANGE vymazat. Aby poslední řádek v této tabulce zodpovídal stavu ve kterém se skutečně nacházíme. Když však tyto řádky skutečně vymažeme, ztratíme možná cennou informaci. Proč je tedy taky jenom nedeaktivovat? Taková deaktivace by však z pohledu zevnitř systému byla nezvratná. Tj. Tyto řádky by viděl jenom superdministrátor s přístupem do databáze. Uvedu jako možnost budoucího vývoje, že tyto změny by pak mohli být obhospodařovány v další tabulce, do které by sa zapisovali změny v tabulce CHANGE.

Z důvodu, že se id uživatele ukládá do tabulky změn, musí být i tato a od ní závislé administrátorské tabulky deaktivovatelné (tj. Entity se nemohou maza). Změny v nich však budou ze systému zachytávání změn vyloučeny (není potřeba tyto údaje ucovávat).

### Oživování mrtvých entit.

Systém umožní taky oživení mrtvých entit a to buďto jednotlivě nebo s celým stromem, který z tohto kořenu vyrostl, podle výběru uživatele. Tyto změny se do tabulky CHNGE zapisovat budou.

## E.4 návrh mapy stránek

Jsou zobrazeny podle typu uživatelů. Vždy platí, že vyšší role zahrnuje, nebo překrývá stránky nižší role.

## Mapa stránek uživatele občan.

### View 1. Přihlášení:

Netýká se role občan.

### View 2. Vstupní stránka:

Vstup: View 1, nebo nic, resp. každý pokus o vstup na více privilegovanou stránku.

Výstup: View 3, View 4.

Společná pro všechny role. Na vstupní stránce bude možnost si zvolit zemepisnou oblast, kde sa nachází veřejný orgán(View. 3) , nebo konkrétní osobu(View 4), která nás zajímá.

### View 3. Veřejný orgán(E):

Vstup: view 2,

Výstup: view 2, view 5, view 6

Na této stránce budou zobrazeny základní údaje verějného orgánu (adresa, předseda, seznam aktuálních poslanců, etc..), komponenta veřejných rolí (3.a). a všech hlasování (3.b) daného veŘejného orgánu. resp. sady hlasování. Bude tu taky volba náhledu do historie, po zaškrtnutí které se objeví výběr hasovacích období, resp. kalendář které umožňují náhled do minulosti. Protože se entita volebního období nevztahuje k verejnému orgánu, ale veřejné osobě, je teoreticky možné, že v daném momentu jsou činné veřejné role, které nemají odkaz na stejnou entitu volební období (TENURE). V tomto případě bude k filtrování historie kalendář. Běžnou praxi však je, že volební období se prakticky vždy vztahuje ke veřejnému orgánu a proto ponechám filtrování (2. druhu) i přes výběr volebních období, které je uživatelsky logičtější.

**komponenta 3.a Veřejné role:** V této komponentě budou zobrazeny buďto aktuální sada aktuálních rolí (tj. např. poslanci), nebo po zaškrtnutí historie sady rolí které byli aktuální v jistém momentu historie.

**Komponenta 3.b Hlasování (plurál):** Tato komponenta zobrazí všechna hlasování, která se odehrala na půdě daného veřejného orgánu ve vybraném volebním období. Pokud nebude volební období pro všechny veřejné role daného veř. Orgánu stejné, bude tu vymezeno rozpětí od – do (kalendáře). A podívat se na jejich detailnější zobrazení (View 6.). Zde bude také grafická komponenta zobrazující rozmístnění hlasování v čase.

### View 4. Veřejná osoba(E):

Vstup ze: View 2, view 3, view 5

Výstup na: view 2, view 3, view 5, view 6

Po zvolení veřejné osoby (View 2) budou zobrazeny její základní osobní údaje a možnosti výběru její veřejných rolí (4.a), které zehrála, nebo nahlédnutí do hodnocení této osoby(4.b).

**Komponenta 4.a Veřejné role:** Zde bude možné vybrat konkrétní veřejnou roli, pro zvolené volební období.(View 5.)

**Komponenta 4.b Hodnocení veřejné osoby(E):** Zde bude k nahlédnutí hodnocení veřejné osoby.

### View 5. Veřejná role(E):

Vstup: View 3, view 4

Výstup: view 3, view 4, view 5, View 6

Zde se zobrazí základní údaje dané veřejné role a seznam hlasování ve kterých se daná osoba zúčastnila. Vedle hlasováních bude uvedeno jak se tato role v daném hlasování zachovala.

**(//variant 2: hlasovania bude combo box a po výbere sa oživí komponenta 5.a Komponenta 5.a Hlasování role(E):** //Po vybrání hlasování se zobrazí, jak tato role hlasovala.)

### View 6. Hlasování (E):

Vstup: View 3, view 4, view 5

Výstup: view 3, view 5

Zde se zobrazí podrobnosti vybraného hlasování (tj. výsledku hlasování) + zobrazení kdo jak hlasoval.

**Komponenta 6.a Předmět hlasování(E):** Podrobnosti o předmětu hlasování (k danému předmětu se může hlasovat více-krát).

**Komponenta 6.b Tématický okruh hlasování(E):** Přislouchá předmětu hlasování. Podrobnosti o něm.

**Komponenta 6.c Hlasování osob:** Zde se zobrazí kdo jak hlasoval.

**Komponenta 6.d Klasifikace Hlasování:** Zobrazení ohodnocení hlasování.

Ve všech stránkách / komponentách které zodpovídají nějaké entitě, bude komponenta zobrazující dokumenty přislouchající této entitě ke stažení.

### Mapa stránek dobrovolníka.

Změněné stránky: Na každé stránce, která představuje entitu (označení E) přibudne tlačítko: „přidej novou entitu“ a komponenta dokumentů bude editovatelná, tj. bude možnost přidávat/odebírat dokumenty. Přibudne stránka (View 7.), na které budou odkazy na univerzální správcovké rozhraní pro všechny entity. Samotný editovací stránka (view 8) bude univerzální, tj. prispůsobena všem typům entit. Umožní jak editaci, tak přidávání i mazaní entit tříd týkajících se business modelu a editace vlastního profilu (hesla a pod).

### Mapa stránek administrátora.

Všechno jak pro dobrovolníka. Editace i administračních tabulek, přidávaní a mazání nových uživatelů. Přístup k možnosti navrácení systému.

## E.5 Možnosti rozšiřování systému.

**Problém vyššího počtu dobrovolnických skupin.** V případě, že se systém rozšíří, bude třeba odseparovat různé dobrovolnické skupiny, aby si „nekafrali do zelí“. Řekněme, že budou 2 dobrovolnícke skupiny, monitorující chování městkých zastupitelství v Košicích a Bratislavě. Jak by třeba vypadalo návrat do minulosti? Co by tomu řekli lidé z druhé skupiny?

Tento problém je možné řešit zahrnutím veřejného orgánu do přihlášení (je pravděpodobné, že dobrovolníci z jedné skupiny budou „obsluhovat“ jenom jeden verejný orgán) a daný dobrovolník bude mít dosah jenom na entity příslušící danému orgánu. To samé platí pro administrátora jak jsme si jej definovali výše. Jeden administrátor by měl připadat na jednu skupinu dobrovolníků, ideálně pro jeden veřejný orgán. Nicméně musí tady být potom vytvořena další role, řekněme „super-admin“, která umožní přidávání entit veřejných orgánů a tvorbu kont lokálních administrátorů. Tady by zase platilo, 1 super-amin na jednu databázi.

**Řešení 1.** Všechny tyto argumenty hovoří v prospěch oddělených databází. To by však mělo za nevýhodu to, že pokud daná osoba působila v různych veřejných orgánech, její existence v systému bude vícnásobná a tyto budou vzájemně nezávislé. Pak v jedné společné databázi(D0) budou tabulky T_KRAJ, T_OKRES, T_LOCATION, T_PUBLIC_BODY a T_DATABASES_CONNECTION z těchto tabulek nebude možné mazat. V lokálních databázích(D_loc) pak zase budou tabulky od T_PUBLIC_BODY včetně všechny ostatní a tabulka T_PUBLIC_BODY bude obsahovat právě jeden záznam. Při vložení nového záznamu do T_PUBLIC_BODY v D0 se spustí script na vytvoření nové databáze (název nové DB, login a heslo se uloží do T_DATABASE_CONNECTION), tabulkové schéma, inicializuje nového administrítora (tj. např „admin“/“admin“), a vloží 1 (jediný) údaj do T_PUBLIC_BODY. V Java světě se

budou udržovat 2 databázová spojení do D0 i D_loc, přičemž D_loc se operativně nastaví při vstupu na stránky (uživatel si bude muset vybrat veřejný orgán, kam bude chtít vstoupit). Do D0 bude mít přístup jen super-admin.

**Řešení 2.** Budeme u dobrovolníku předpokládat „dobrou vůli“, tj. že nebudou vědomě zasahovat do cizích záležitostí. Do tabulky A_CHANGE pak přidáme stloupec „public_body_id“, který bude evidovat příslušnost k veřejnému orgánu. Návraty do minulosti pak budu zohledňovat jenom řádky, které se týkají našeho veřejného orgánu (tj. oblasti, kde pracují dobrovolníci). Vstup do systému bude opět podmíněn výběrem veřejného orgánu (kvůli znížení pravděpodobnosti vzniku chyb. Kdyby totiž dobrovolníci pracovali na různých veřejných orgánech současně, a chtěli se vrátit do minulosti, nesměli by zapomenout návrat skrz všechny veřejné orgány, na kterých by pracovali, takhle to bude pro ně více zřejmé, protože se budou muset přepnout).

## F. Fyzické umístnění

Aplikace bude pracovat jako non-stop webová služba na aplikačním serveru *Tomcat*. Server (hardware), kde je aplikace umístěna musí podporovat taky databázový server *MySQL*. Prozatím se předpokládá pronájem místa. Později, pokud bude aplikace úspěšná zakoupení vlastního serveru (hardware).

## G. Zajištění kvality

Uživatelé budou moci, pokud zaznamenají funkční chybu kontaktovat IT support a tuto jim sdělit. Jedná se o klasický post-launch support.

## H. Závěr

Nepovažuji tuto práci za zdaleka ukončenou.  Některé části, kterými se budu zaobírat, jsem již výše nastínil a mohé další na mne jistě čekají v záloze.