

Obsah

| | |
|--|----|
| A. Cíle aplikace..... | 1 |
| Cíle-řešení: | 3 |
| Úroveň technických detailů: | 4 |
| Priorita cílů (požadavků)..... | 5 |
| B. Uživatelé a lidský faktor..... | 7 |
| C. Návrh řešení..... | 8 |
| Ukládání dat..... | 8 |
| Uchovávání vnitřní historie, tj. historie akcí apod..... | 8 |
| Spravovatelnost systému..... | 9 |
| Přihlašování..... | 10 |
| Návrh databázového modelu | 10 |
| Entity týkající se bussiness modelu: | 10 |
| Pomocné entity, resp. týkající se administrace:..... | 10 |
| Pomocné entity..... | 14 |
| Dodatek..... | 14 |
| D. Implementace, použité technologie..... | 18 |
| Maven..... | 18 |
| Reflexe | 18 |
| Vaadin | 19 |
| Trochu historie..... | 21 |
| Architektura Vaadinu | 21 |
| Technologické pozadí, HTML a Javascript, CSS..... | 22 |
| AJAX..... | 23 |
| Psaní aplikací pro serverovou stranu | 23 |
| Základní prvky aplikací UI..... | 24 |
| Instalace aplikace („deployment“)..... | 25 |
| E. Technické detaily řešení | 27 |
| Řešení přístupu do databáze..... | 27 |
| Filozofie connections a transakčních operací. | 28 |

| | |
|--|----|
| Odstraňování / deaktivace entit | 29 |
| Filtrování..... | 31 |
| Ukládání změn | 31 |
| Univerzální formát..... | 32 |
| Ukládání dokumentů..... | 33 |
| Návrh mapy stránek..... | 33 |
| Mapa stránek běžného uživatele - občana | 33 |
| Mapa stránek dobrovolníka..... | 44 |
| Mapa stránek administrátora. | 47 |
| Možnosti rozšiřování systému..... | 48 |
| Grafická podoba stránek. | 48 |
| Návrat do bodu v minulosti | 48 |
| Oživování mrtvých entit. | 49 |
| Problém vyššího počtu dobrovolnických skupin. | 49 |
| F. Fyzické umístění | 51 |
| G. Testování | 51 |
| H. Zajištění kvality..... | 51 |
| I. Závěr..... | 52 |
| J. Epilóg..... | 55 |
| Seznam Literatury | 57 |
| Přílohy | 58 |
| Uživatelský manuál..... | 58 |
| A. Instalace | 58 |
| B. Příprava na straně aplikace. | 59 |
| C. Spouštění aplikace | 59 |
| D. TroubleShooting..... | 60 |

Abstract

This thesis is about creating web application for monitoring representatives voting behavior by using framework *Vaadin*. Application will be reachable 24/7 for all people. Some of them, with superior authority will be allowed to insert/edit system important information (entities, comments and documents). The scope of the thesis is not to create completely finished application, but its basic functional skeleton.

Abstrakt

Tato práce se zabývala tvorbou webové aplikace na monitorování činnosti veřejných činitelů s využitím frameworku *Vaadin*. Aplikace bude veřejně přístupná pro běžné uživatele. V aplikaci budou mít uživatelé s vyššími právy možnost do systému vkládat/editovat entity, komentáře a dokumenty. Rozsah této práce nepokrývá tvorbu celé aplikace, jen její základní funkční části.

A. Cíle aplikace

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

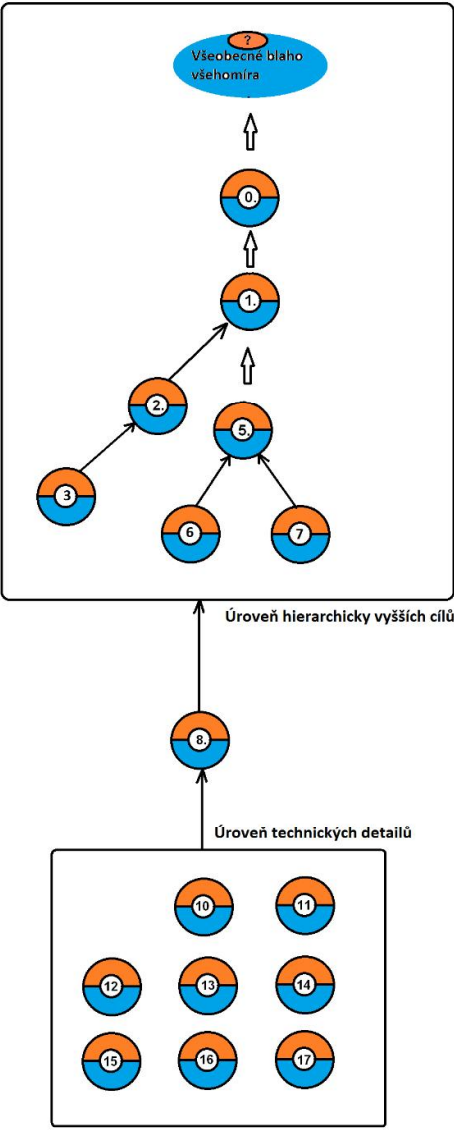
Cílů může být tolik, kolik je úhlů pohledů. Spíš než o cílech jednotlivě bude lépe, když budeme mluvit o *systému cílů*, který má svou vnitřní strukturu a hierarchii. Následující očíslované body se náš *systém cílů* pokusí načrtnout, nicméně z povahy věci plyne, že tento nebude úplně vyčerpán.

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 dosažení 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 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.

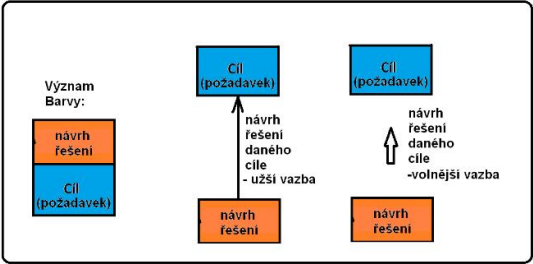
Graficky je *systém cílů* našeho systému znázorněn na obrázku č.1. Šipky na obrázku je třeba chápat víceméně symbolicky, protože cíle se ve skutečnosti vzájemně překrývají a naopak, dané řešení může sloužit k dosažení mnoha cílů. Nicméně je možné si všimnout, že tyto cíle/ řešení je možné smysluplně 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



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. výkonu její 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. nemyšlení na něj) tak vede k celkové nepřehlednosti systému. V takovéto zbytečné nepřehlednosti (tj. Nepřehlednosti která nevyplývá 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. toto 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 dobrovolníkovi vkládání komentářů k událostem. |
| priorita: | L1-vysoká |

| | |
|------------------|--|
| č. požadavku: | 14 |
| název: | Správa entit (objektů) |
| uživatel: | dobrovolník, administrátor |
| popis: | Aplikace umožní správu (vytváření/naplňování daty/mazání) entit (objektů), které jsou popsány v DB modelu. |
| priorita: | L1-vysoká |

| | |
|------------------|--|
| č. požadavku: | 15 |
| název: | prohlížení profilu veřejných osob, etc. |
| uživatel: | Všichni |
| popis: | Systém bude generovat přehled údajů a historie působení jednotlivých veřejných činitelů / veřejných orgánů / a ostatních entit Tj. LOCATION, VOTE, a jiné... |
| priorita: | L1-vysoká |

| | |
|------------------|--|
| č. požadavku: | 16 |
| název: | Trojí úroveň zobrazování |
| uživatel: | všichni |
| popis: | Aplikace bude podporovat trojí úroveň 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, nabízí se opět pyramida, tentokrát z oboru psychologie - *Maslowová pyramida lidských potřeb*. Potřeby, které jsou u základny pyramidy, jsou sice pro život naprosto nevyhnutelné, nicméně na směřování 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 může - nač však bude funkční, když ji nikdo nebude používat? (L2)

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

B. Uživatelé a lidský faktor

System 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). System 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 docela 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 zadána, 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 byl co nejprehlednější, snadno udržitelný 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), který vychází z modelu MVC (C=controller).

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

„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 projevíly do změn ve „View“, který představuje zobrazovací i ovládací prvek, většinou s výstupem na monitor.

Ve všeobecnosti možno hovořit o MVP v případě „těžkých“ View, tj. kdy stav View závisí na stavu Modelu netriviálně a kde nelze jednoznačně rozlišit mezi vstupem a výstupem. Ukázkovým případem jsou Web stránky, kde tlačítko může sloužit jak na výstup (objeví se, nebo zmizí) i jako vstup (stlačením dává uživatel pokyn). Naopak MVC je aplikovatelný v případě „lehkých“ View, kde jsou vstupy a výstupy striktně odděleny. Jako příklad MVC bych uvedl fyzický knoflík jako vstup a diodu jako výstup – indikuje změnu stavu po stlačení knoflíku 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 databáze. Opět, jelikož ukládaná data nejsou zvlášť choulostivá, 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 během svého života může vykonávat vícero veřejných funkcí, čili rolí.

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

Představuje konkrétní veřejnou 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 přislouchat všem veřejným funkcím 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. 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 jedno hlasování veřejného orgánu. Některé atributy VOTE, jako např. počet hlasů Pro / Proti / atd. resp. celkový výsledek se můžou jevit jako nadbytečné, protože informaci v nich uloženou možno dopočítat z výsledků hlasování jednotlivých funkcí (a při tvorbě databáze 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 odkazovat 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 tématu. 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 vícerych krocích hlasování - tj. kauza, nebo všeobecná témata například: územní rozvoj, vzdělávání, atd.

10. VOTE_CLASSIFICATION (hodnocení hlasování)

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

Kritéria hodnocení musí být taková, aby v co nejmenší míře mohla být považovaná za subjektivní a nemělo by jich být mnoho. Mohla by se hodnotit na škále 0 – 5, např. „*veřejná prospěšnost*“.

Každé hodnocení musí být podloženo odkazy na fakta(dokument, nebo poznámku), která k tomuto hodnocení vedla.

Kvůli zvýšení objektivit vyhodnocování bych navrhoval, aby do úvahy přicházela výhradně fakta (tj. návrhy hlasování a hlasování) uložená v modelu. Tedy model bude zachovávat presumpci nevinoty 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 tato 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 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 se budou evidovat všechna vložení od daného uživatele. Další podrobnosti v kapitole *Implementační detaily*.

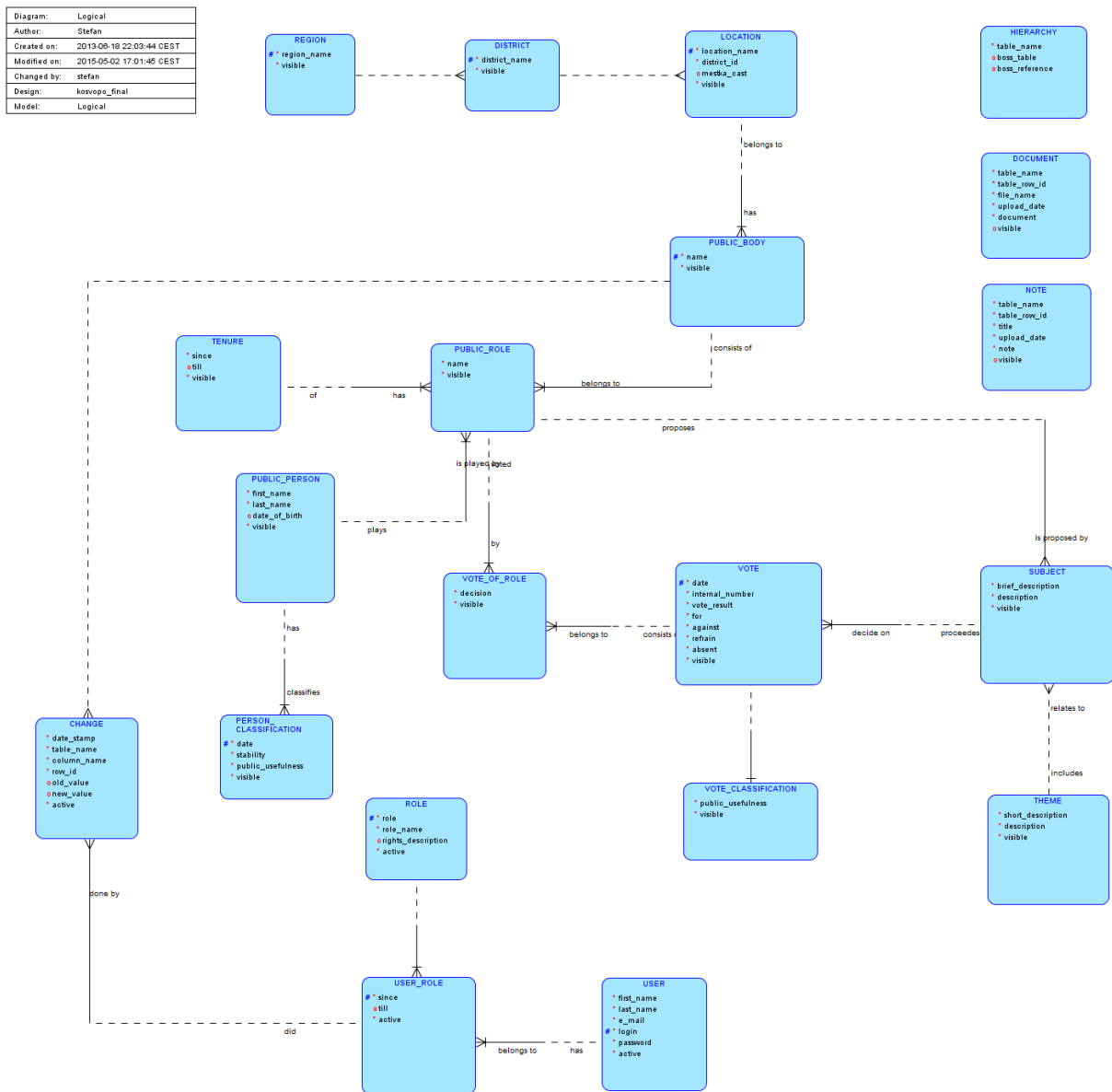
18. HIERARCHY (hierarchie)

Entita mapující hierarchické vztahy mezi entitami *bussiness* modelu. Další podrobnosti v kapitole *Implementační detaily*.

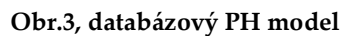
Dodatek

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 se 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.



Obr.2, databázový ER model



D. Implementace, použité technologie

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. [1] Hlavním impulzem pro jeho výběr v projektu je jeho univerzálnost a rozšířenost.

Reflexe

Reflexe je schopnost programovacího jazyka zjistit za běhu informace o určitém objektu. Obecně, poněvadž 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)

Vaadin

Tenhle framework jsem si vybral, protože byl právě pro tento typ aplikací navržen. Konkrétně pro následovní 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 vědom i jeho nevýhod, na které jsem počas vývoje aplikace narazil:

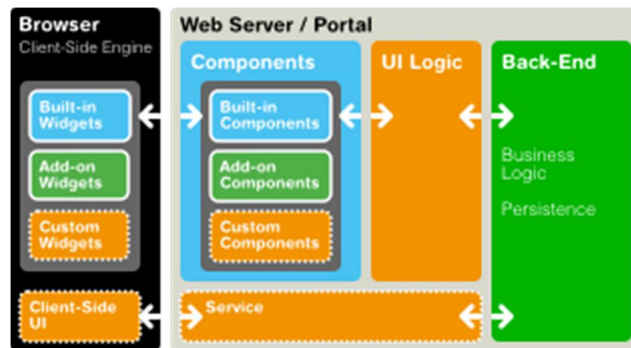
- Ukrytování technologií, na kterých systém běží před zraky programátora. Tj. všechno je perfektní až do doby, když to přestane fungovat a hlavní výhoda se zvrátí v hlavní nevýhodu.
- Hotová řešení se někdy těžko upravují do podoby, kterou zrovna vyžaduje situace.

Co je to Vaadin? Vaadin je „Java web developement Framework“ určený k 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 překladač 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 z 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. [4]

Vaadin se tak stará o obsluhu uživatelského rozhraní a AJAX-ovou komunikaci mezi browserem a serverem. Viz. Obrázek 4.



Obr.4, Schéma komunikace mezi serverem a browserem ve Vaadinu.

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. [4]

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..). [4]

Vaadin je závislý na podpoře 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, který poskytuje compiler z Javy do JavaScriptu, který běží na prohlížeči. Tedy tak nebo tak developer přichází do kontaktu jedině s Javou. [4]

Vaadin taky podporuje jasné oddělení mezi strukturou uživatelského rozhraní a jeho vzhledem, a umožňuje je vyvíjet separátně, nezávisle na sobě. Vaadin řídí vzhled stránek

kompletně skrz tzv. *témata*, která využívají CSS dokumenty, případně šablony HTML stránek. [4]

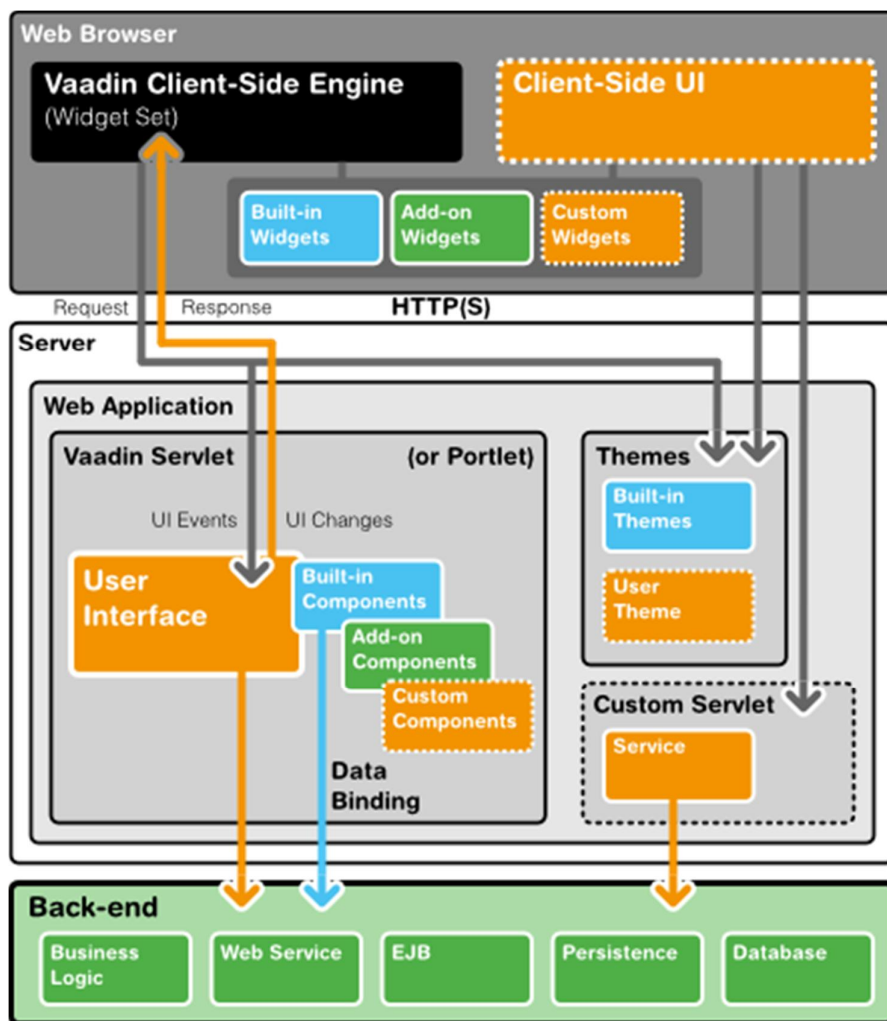
Základní třídou, ze které se ve Vaadinu vychází je třída, která dědí z třídy *com.vaadin.ui.UI*. UI je částí webovské stránky, ve které Vaadinovská aplikace běží. UI je sdružená se uživatelským *session*, které je vytvořeno pro každého uživatele pracujícího s aplikací. Základní metodou této třídy je *init()*, která se spouští automaticky při prvním vstupu do aplikace. [4]

Trochu historie

Vaadin je víceméně produktem plynulého vývoje myšlenky vývojového prostředí pro tvorbu webových aplikací. Řekněme, že tento vývoj začal v roce 2000 produktem skupiny IT Mill. Tehdy měl jeho předchůdce název Millstone library. Tato měla 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 byla kompletně přepsána pomocí GWT, Google Web Toolkit. Od tohoto 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. [4]

Architektura Vaadinu

Následující obrázek č.5 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. [4]



Obr.5, Architektura Vaadinu.

Serverová část aplikace běží jako servlet na Java web serveru, obsluhujícím http žádosti. Obvykle 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 stránky, kterou uživatel zrovna prohlíží. [4]

Technologické pozadí, HTML a Javascript, CSS

Téměř celý web je postaven na technologiích HTML, který definuje strukturu stránky. Definuje 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 zkompileovány právě do JavaScriptu, pomocí Vaadin Klient Compiler-u. [4]

Vaadin zhusta skrývá použití HTML, dovoluující 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. [4]

Z webových technologií je převzata 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, z designového hlediska docela propracovaná 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. [4]

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. [4]

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 vytvořena. [4]

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. [4]

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()`. [4]

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. [4]

Navigace

Vaadinovské aplikace nemají navigaci jaká je běžná u normálních stránek. Tyto běžně běží na jenom 1 stránce podobně jako všechny aplikace založené na Ajaxu. Běžné je, že aplikace mají vícero View, mezi kterými uživatel přepíná. Tuto službu dělá ve Vaadinu třída `Navigator`. Views spravované pomocí něho automaticky získávají specifické URI, které může být použito na přímou navigaci, skrz příkazový řádek v browseru. [4]

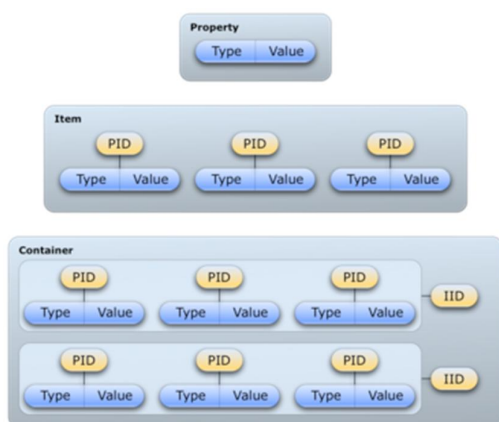
Komponenty uživatelského rozhraní

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`, `TextField`, `ComboBox`, `Window`, atd... Uživatel – programátor si může na základě existujících komponent sám vytvářet nové komponenty. [4]

Vázání komponent a dat(Data binding)

Jednou z filosofí 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 nevyklučuje, 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 odpovídají buňce, řádku resp. celé databázové tabulce. [4]

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. *Container*, *Property* a *item* se můžou přímo vázat ke komponentám, přičemž *container* ke komplikovanějším (Table, ComboBox), *property*, resp. *item* zase k jednodušším (TextField). Vztah mezi *Item*, *property* a *container-em* je znázorněn na dalším obrázku.



Obr.6, Schéma ukládání dat ve Vaadinu.

Instalace aplikace („deployment“)

Vaadinovské aplikace jsou instalovány jako Java webovské aplikace, na webovém serveru (Tomcat, GlassFish, aj.). 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`). Podobně jako obyčejný Jar soubor je War ZIP-komprimovaný, se specifickou strukturou obsahu. [4]

V hantýrce Java – Servletů *webová aplikace* značí kolekce *servletů* a *portletů*, JSP a statických HTML souborů a různých jiných zdrojových souborů, které tvoří aplikaci. [4]

E. Technické detaily řešení

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. Jsem 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 nepřímou 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 rutinní 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 ve 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 dva základní přístupy, jak se s ní vyrovnat.

První přístup vede k „instantnímu přizpůsobení“ tj. jakémusi zrychlenému ztotožnění se s 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 práce 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).

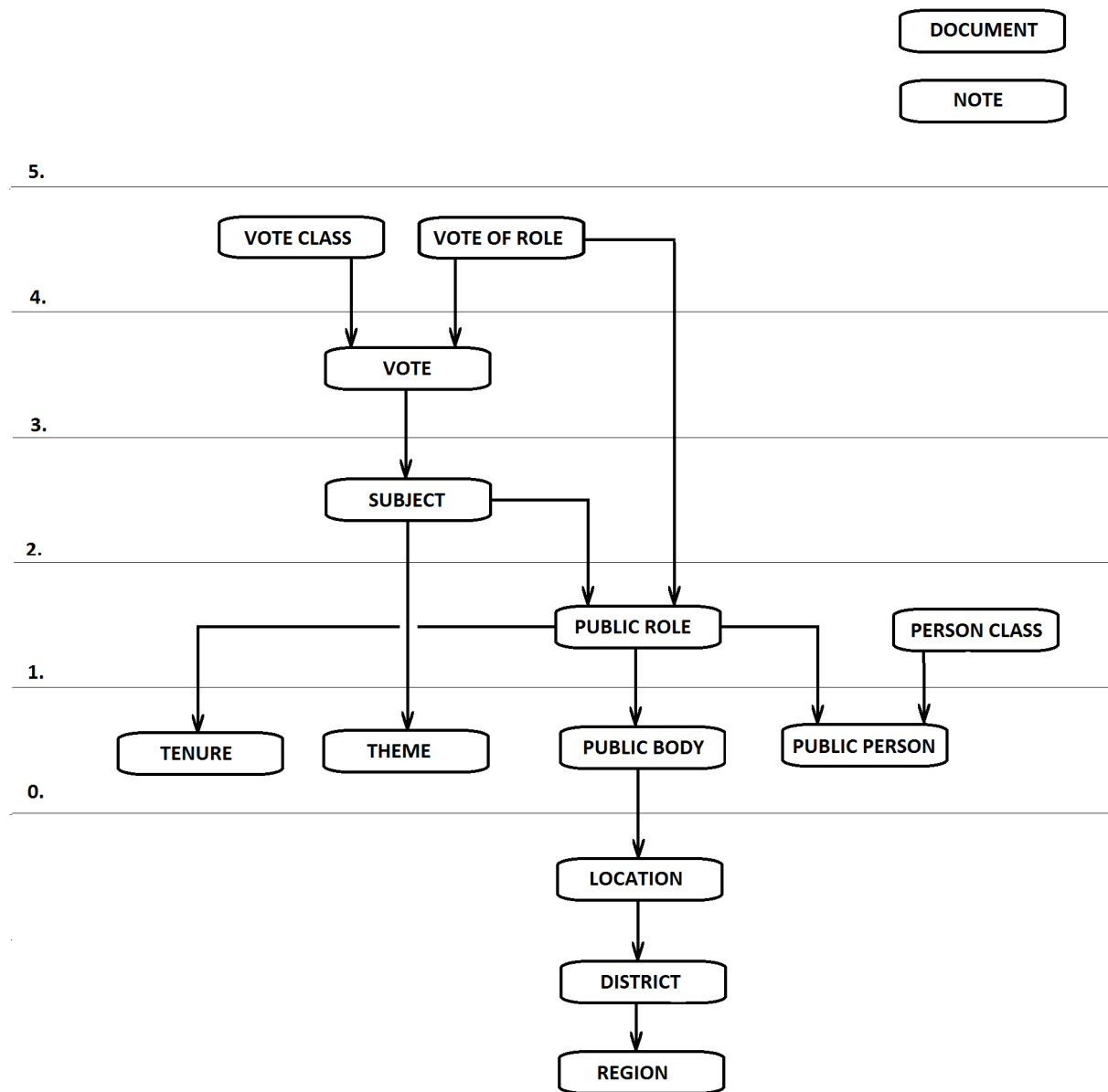
Těto typům operací bude přizpůsobena i správa databázových připojení.

- 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.

Odstraňování / deaktivace entit

Jelikož entity nejsou na sobě nezávislé, ale mají určité vazby, bude nevyhnutelné 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 níže uvedeného filtrování.

Hierarchické vztahy



Obr.7, Strom hierarchických vazeb mezi entitami navrhovaného modelu.

Na obrázku č.7 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 níže 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 měly zmizet?

Algoritmus jejich vyhledávání při deaktivaci bude následující:

1. Vyjdi z entity, kterou chceš 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 popsany v tom samém 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 onen zákon nezbytnosti akcí, vychází z hlubšího zákona "šetření energie", který je mu nadřazený. Pokud nezdeaktivujeme entity, které by stejně systém neviděl, musíme vydávat průběžně energii na to, aby se dalším vývojem aplikace nestaly tyto jiným způsobem přístupné. Tj. stále musíme zbytečně čelit této "hrozbě".

Za druhé, 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 databáze získá větší přehled o tom, co je v jakém stavu. Tj. zvýší se transparentnost celého systému.

Vazby mezi entitami jsou dané samotnou definicí v databázi (cizí klíče). Pro urychlení však bude dobré si vytvořit strukturu, 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).

Filtrování

Při vkládání nových entit do systému (resp. prohlížení entit v administrátorském view-pohledu, viz. View 8 níže) je dobré, aby uživatel měl možnost výběru jejích parametrů zouženu podle vlastních potřeb. Například pokud vkládá nové hlasování ve veřejném orgánu, není vhodné, aby měl na výběr všechny předměty hlasování přístupné v celém systému, ale jenom ty, které vznikli na půdě toho samého veřejného orgánu.

Při konstrukci těchto filtrů využijeme stejný algoritmus jako při deaktivaci entit. S tím rozdílem, že vyhledávání nepůjde od vybrané ovlivňující entity až na úroveň listů, ale jen do takové hloubky, než zarazí na třídu ovlivněné entity. Výhodou tohoto algoritmu je, že je univerzální, tj. umožní jakoukoli kombinaci: ovlivňující vs. ovlivněná třída.

Ukládání změn

Při každé změně systému se tato zapíše do databáze do speciální 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 tohoto způsobu 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 porovnání s prostým vymezením sfér vlivu. Vymezení autonomních oblastí s pouze jedním „pánem“ bývá jednodušší, přehlednější a tím pádem více odolné vůči chybám.

Většina logiky se však řeší na straně Javy. Pokud by se část logiky přesunula na stranu DB, nutně by vznikly „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 zbytečně nezatěžovala tato tabulka). To se týká u nás jediné 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 mohly být ukládány v jednom sloupci, je potřebné najít univerzální formát, na který bude možno transformovat jakoukoli hodnotu a taky co nejjednodušeji provést zpětnou transformaci.

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

Ukládání dokumentů

Dokumenty je možno ukládat v databázi jako typ BLOB, nebo jako odkazy k adresářové struktuře na serveru. Výhodou dokumentů uložených v adresářové struktuře je to, že jsou dostupné i jiným způsobem, a též, že odlehčují databázový provoz. Naopak ukládání dokumentů v databázi se mi jeví jako elegantnější, přenositelnější řešení.

Návrh mapy stránek

Upozornění: V následující kapitole budou sice zobrazeny snapshoty z aplikace, nicméně nejedná se o konečnou grafickou podobu stránek, protože detailní grafické podobě jsem se v rámci této práce nevěnoval.

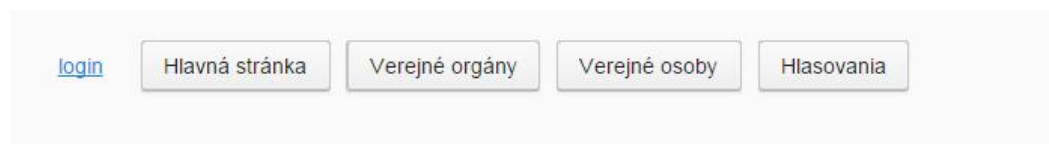
Všeobecné informace: Přihlášení uživatelé budou moci vkládat nov entity, resp. editovat stávající dvěma způsoby.

1. Pomocí tlačítek „*přidej novou entitu*“ a „*edituj novou entitu*“, které budou umístěny na stránkách jen po přihlášení se do systému. Tlačítko *přidej* bude vždy na stránce kde je zobrazen seznam entit daného druhu (v návrhu označeno písmenem *s*, např. *View 4s*). Tlačítko *edituj* pak na stránce zasvěcené dané entitě (*E*).
2. Druhou možností pro vkládání / editaci bude speciální stránka, viz níže *View 8*.

Dále popíšu jednotlivé stránky a jejich funkcionalitu. Vždy platí, že vyšší role zahrnuje, nebo překrývá stránky nižší role.

Mapa stránek běžného uživatele - občana.

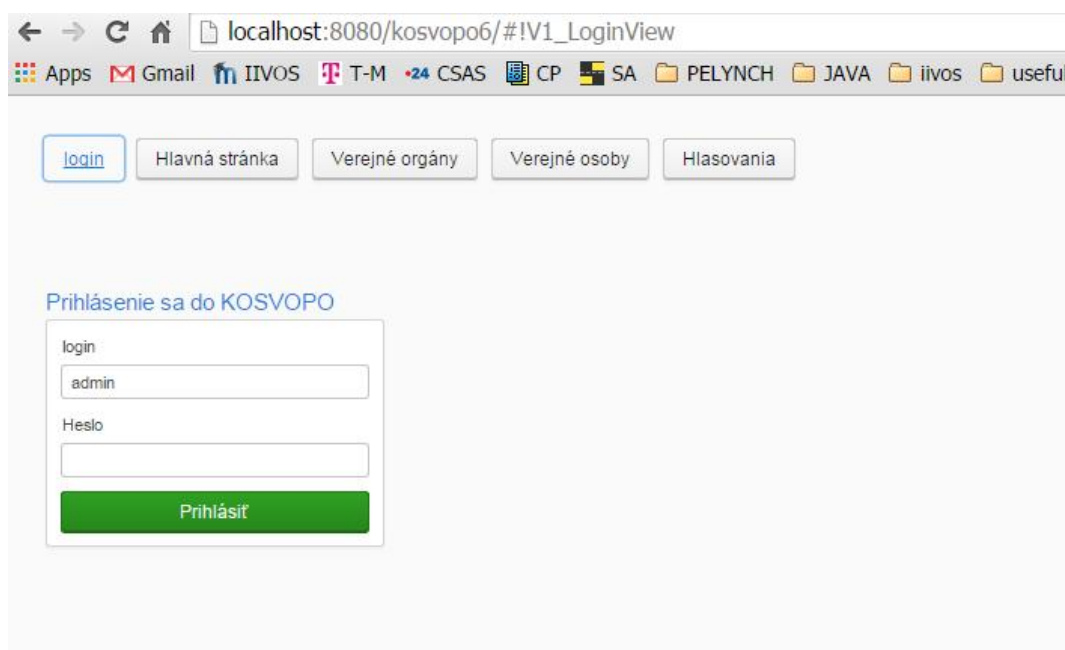
Společnou komponentu pro všechny stránky tvoří odkazy na přihlášení, seznam veřejných orgánů, seznam veřejných osob a seznam všech hlasování.



Obr. 8, Hlavní menu, bez přihlášení.

View 1. Přihlášení

Netýká se role občan, jen role administrátor a dobrovolník.



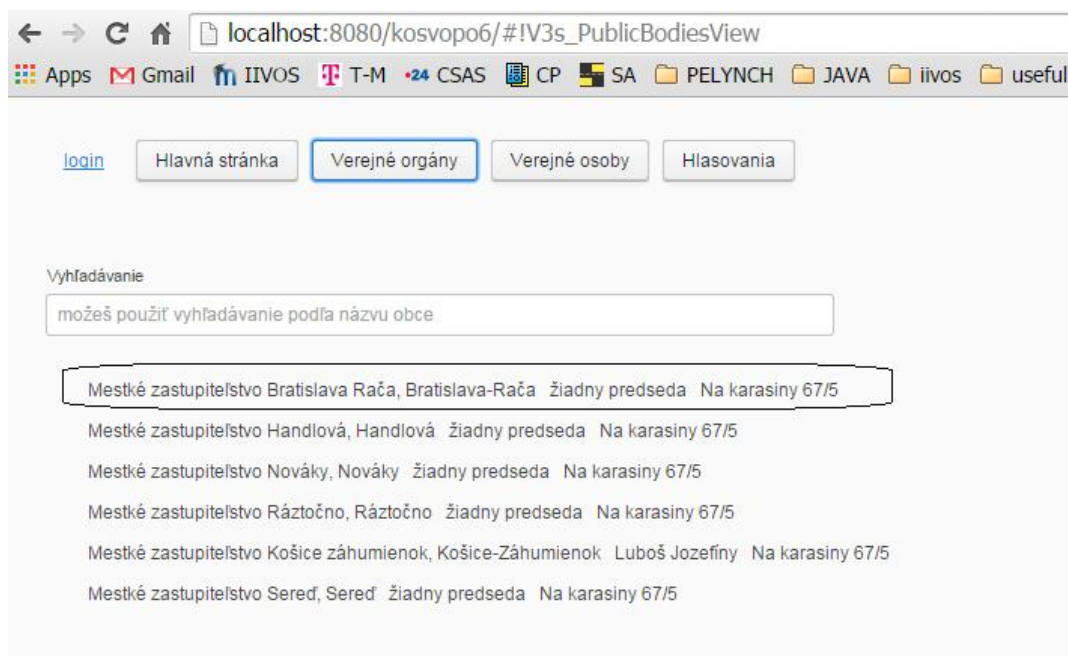
Obr. 9, View 1 - Přihlášení

View 2. Vstupní stránka:

Společná pro všechny role. Zde bude nějaký vstupní nápis, resp. oznámení o co se jedná.

View 3s. Veřejné orgány:

Zde bude seznam všech veřejných orgánů v systému spolu s vyhledávacím řádkem.



Obr. 10, View 3s – Verejné orgány

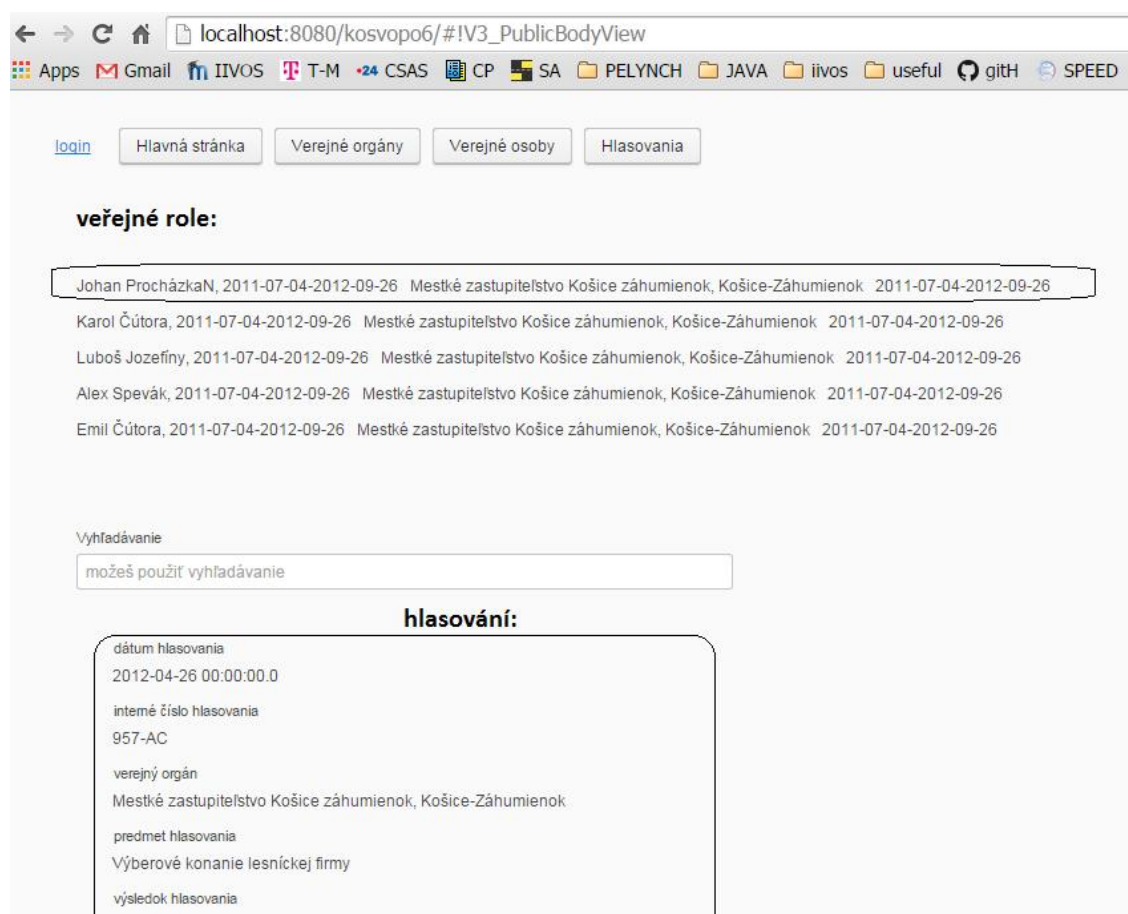
View 3. Veřejný orgán(E):

Vstup: View 3s

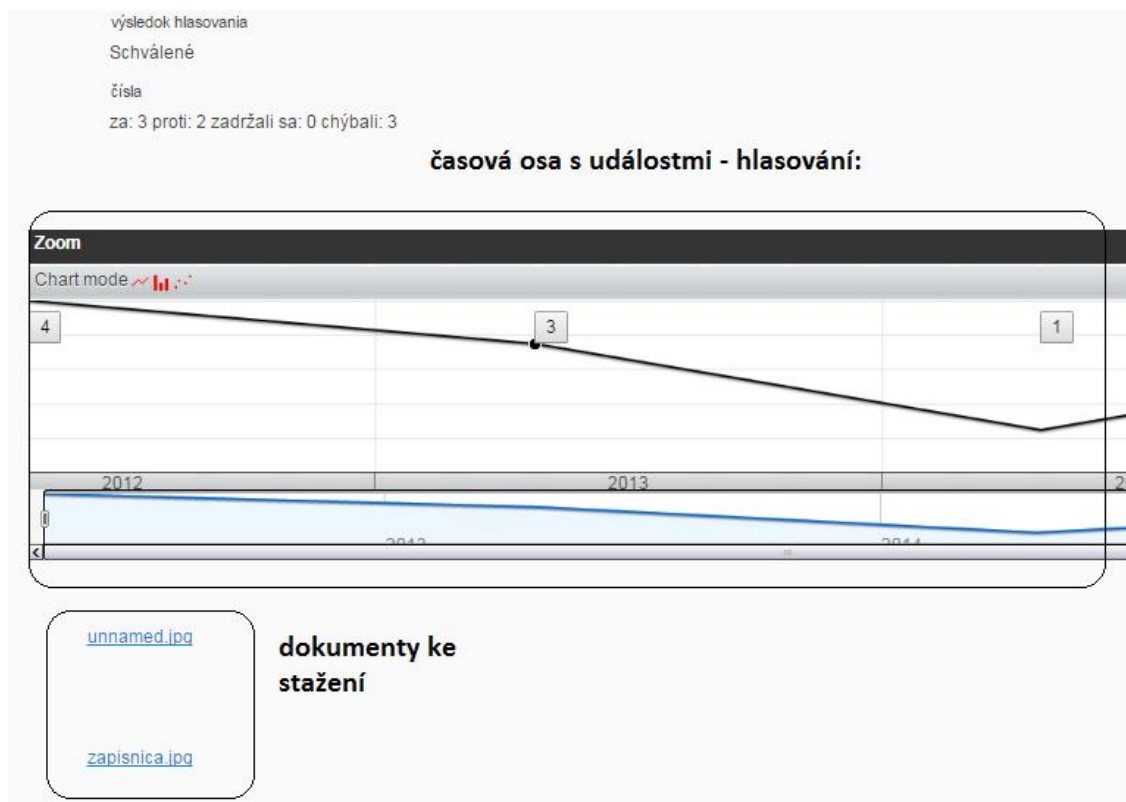
Na této stránce budou zobrazeny základní údaje veřejného orgánu (adresa, předseda, seznam aktuálních poslanců, etc.), komponenta veřejných rolí (3.a). a všech hlasování (3.b) daného veřejného orgánu. resp. sady hlasování. V budoucnu zde bude taky volba náhledu do historie, po zaškrtnutí které se objeví výběr hasovacích období, resp. kalendář, které umožňují náhled do minulosti. Protože se entita volebního období nevztahuje k veřejnému 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 praxí však je, že volební období se prakticky vždy vztahuje ke veřejnému orgánu a proto ponechám filtrování 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 byly aktuální v jistém momentu historie.

Komponenta 3.b Hlasování (plurál): Tato komponenta zobrazí všechna hlasování, která se odehrála 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.



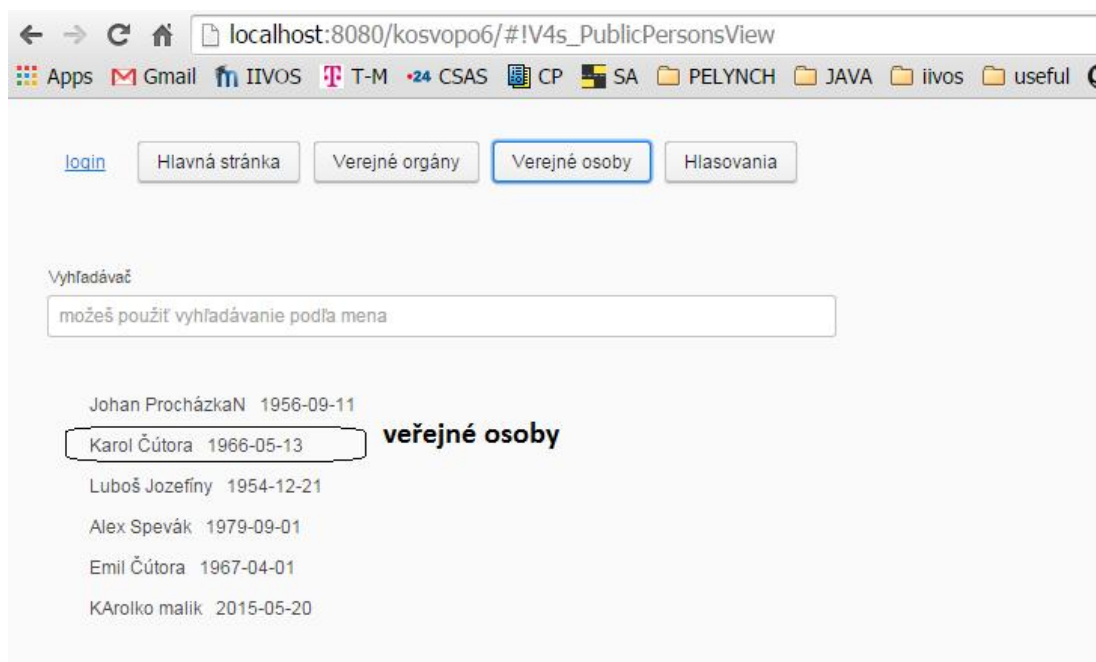
Obr. 11a, View 3- Veřejný orgán, vrchní část



Obr. 11b, View 3- Veřejný orgán, spodní část

View 4s. Veřejné osoby:

Zde bude seznam všech veřejných osob v systému spolu s vyhledávacím řádkem.



Obr. 12, View 4s – Verejné osoby

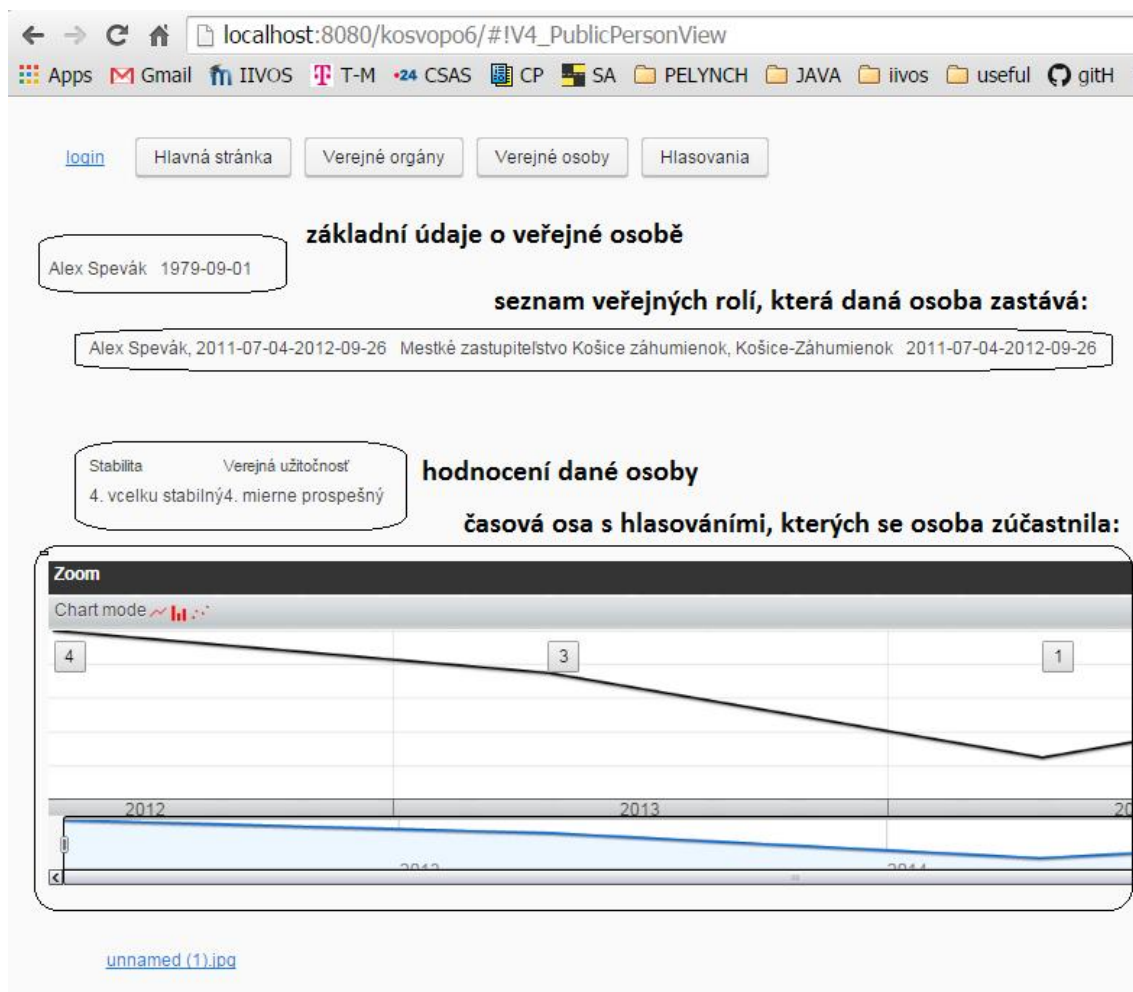
View 4. Veřejná osoba(E):

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

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

Komponenta 4.b Hlasování dané osoby: Zde budou ve formě timeline znázorněny všechna hlasování, kterých se tato osoba kdy zúčastnila.

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

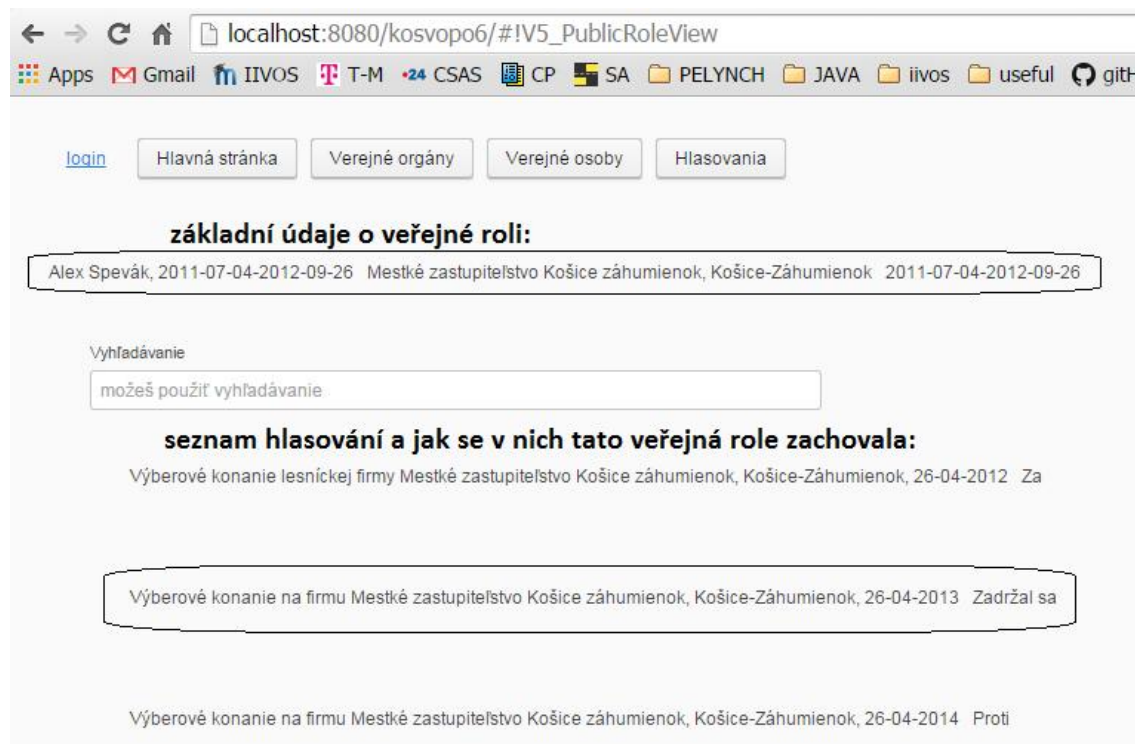


Obr. 13, View 4 – Veřejná osoba

View 5. Veřejná role(E):

Vstup: View 3, View 4

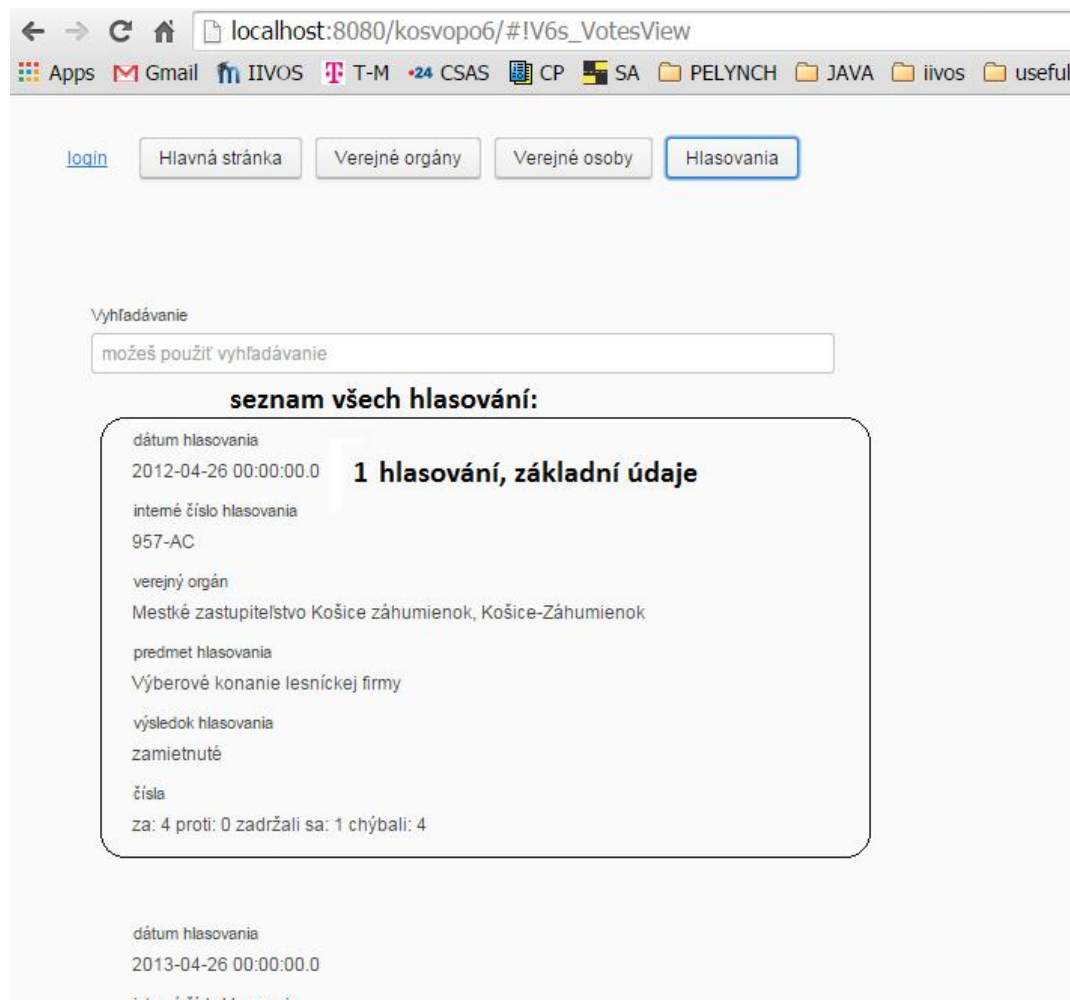
Zde se zobrazí základní údaje dané veřejné role a seznam hlasování, kterých se daná osoba zúčastnila. Vedle hlasování bude uvedeno, jak se tato role v daném hlasování zachovala.



Obr. 14, View 5 – Veřejná role

View 6s. Hlasování (plurál):

Zde bude seznam všech hlasování v systému spolu s vyhledávacím řádkem.



Obr. 15, View 6s – Hlasování(pl)

View 6. Hlasování (E):

Vstup: View 3, View 4, View 5

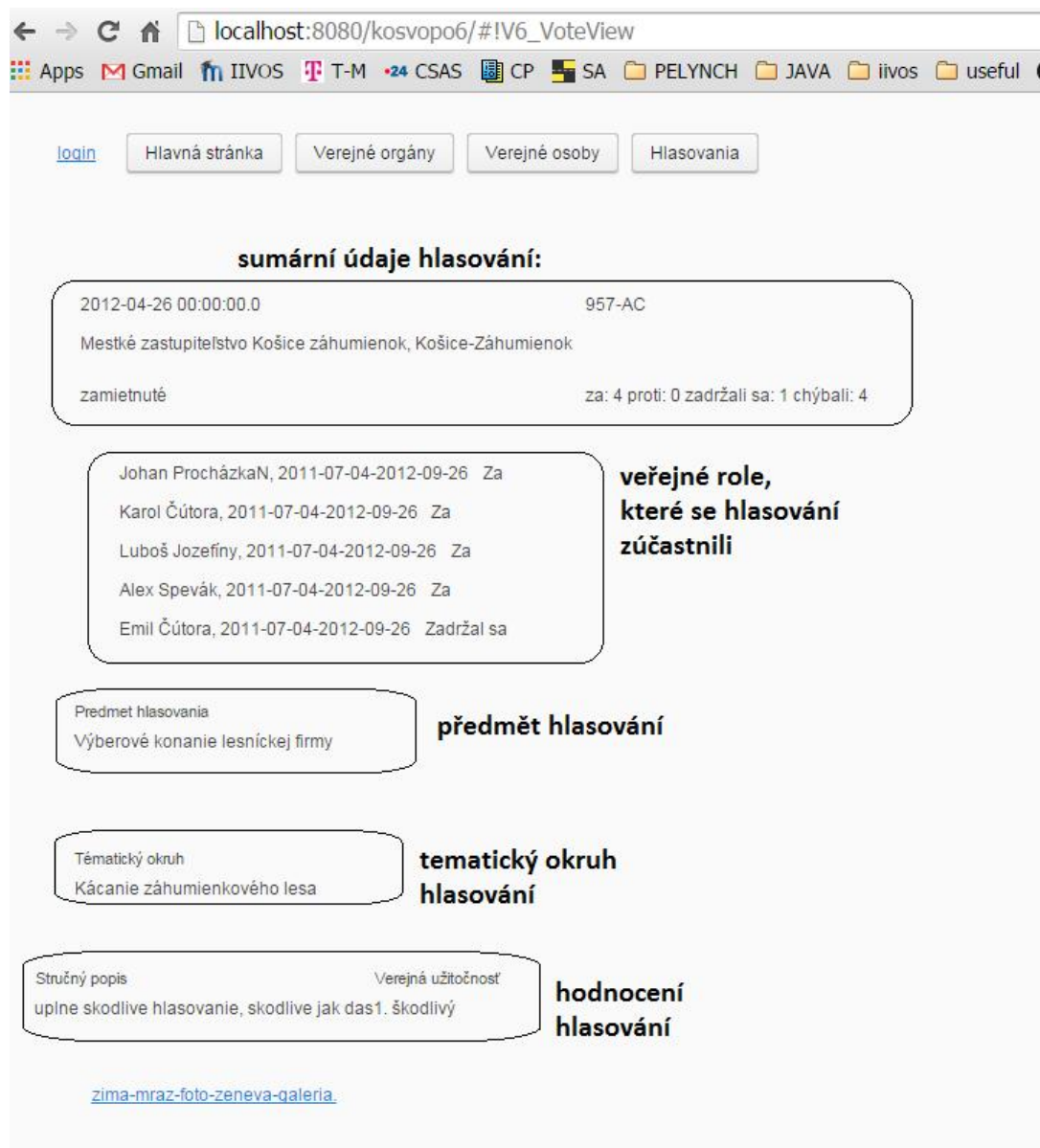
Zde se zobrazí podrobnosti vybraného hlasování (tj. výsledku hlasování) + zobrazení kdo jak hlasoval.

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): Odkazuje na něj předmět hlasování. Podrobnosti o něm.

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

Komponenta 6.d Klasifikace Hlasování: Zobrazení ohodnocení hlasování.



Obr. 16, View 6 – Hlasování (singulár)

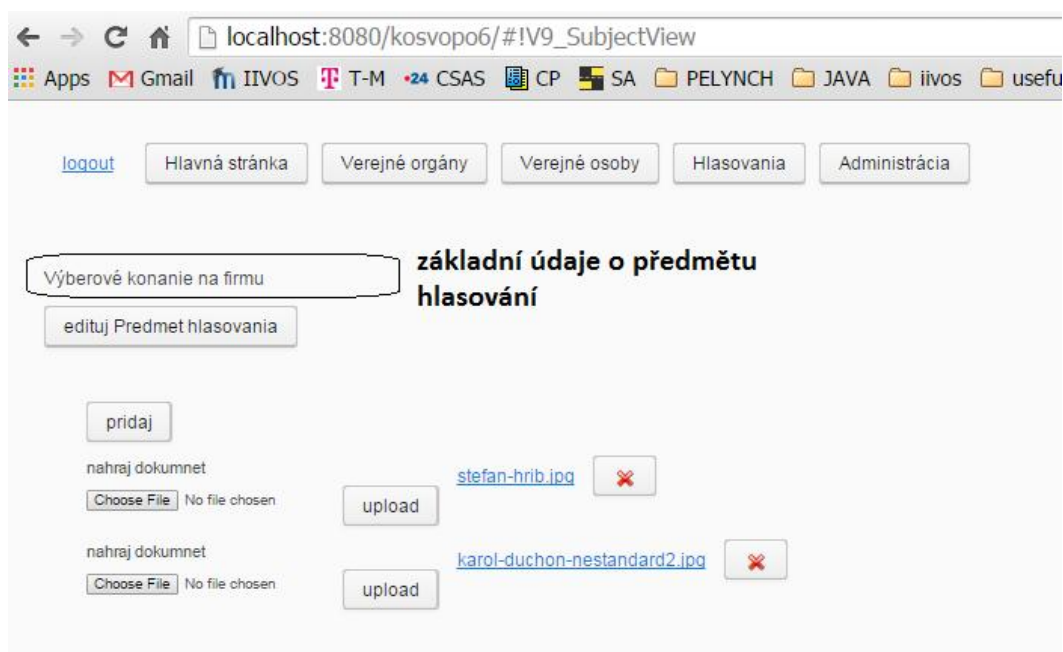
Všechny zde uvedené vnitřní komponenty budou mít listenery a po kliknutí na ně se uživatel přesune na příslušnou stránku.

Ve všech stránkách / komponentách, která odpovídají nějaké entitě, bude komponenta zobrazující dokumenty patřící této entitě ke stažení.

View 9. Předmět hlasování (E):

Vstup: View 6

Zde se zobrazí podrobnosti vybraného předmětu hlasování.

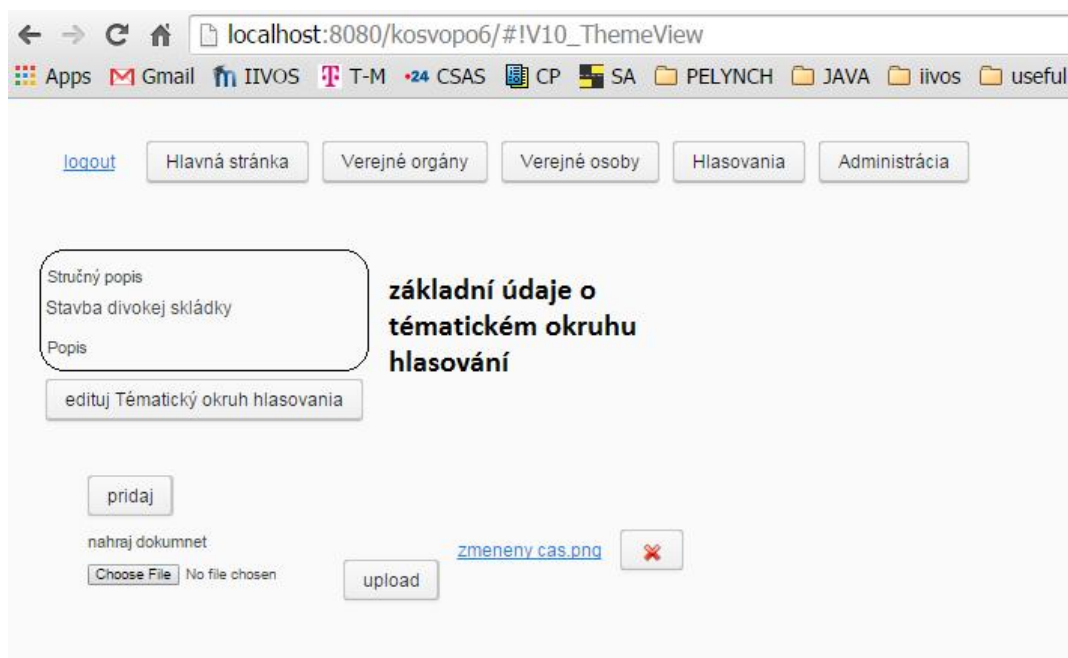


Obr. 17, View 9 – Předmět hlasování

View 10. Tématický okruh hlasování (E):

Vstup: View 6

Zde se zobrazí podrobnosti vybraného tématického okruhu hlasování.



Obr. 18, View 10 – Tématický okruh hlasování

Mapa stránek dobrovolníka.

Změněné stránky: Na každé stránce, která představuje entitu (označení E) přibude tlačítko: „přidej novou entitu“ (s formulářem ve vyskakovacím okně) a komponenta dokumentů bude editovatelná, tj. bude možnost přidávat/odebírat dokumenty.

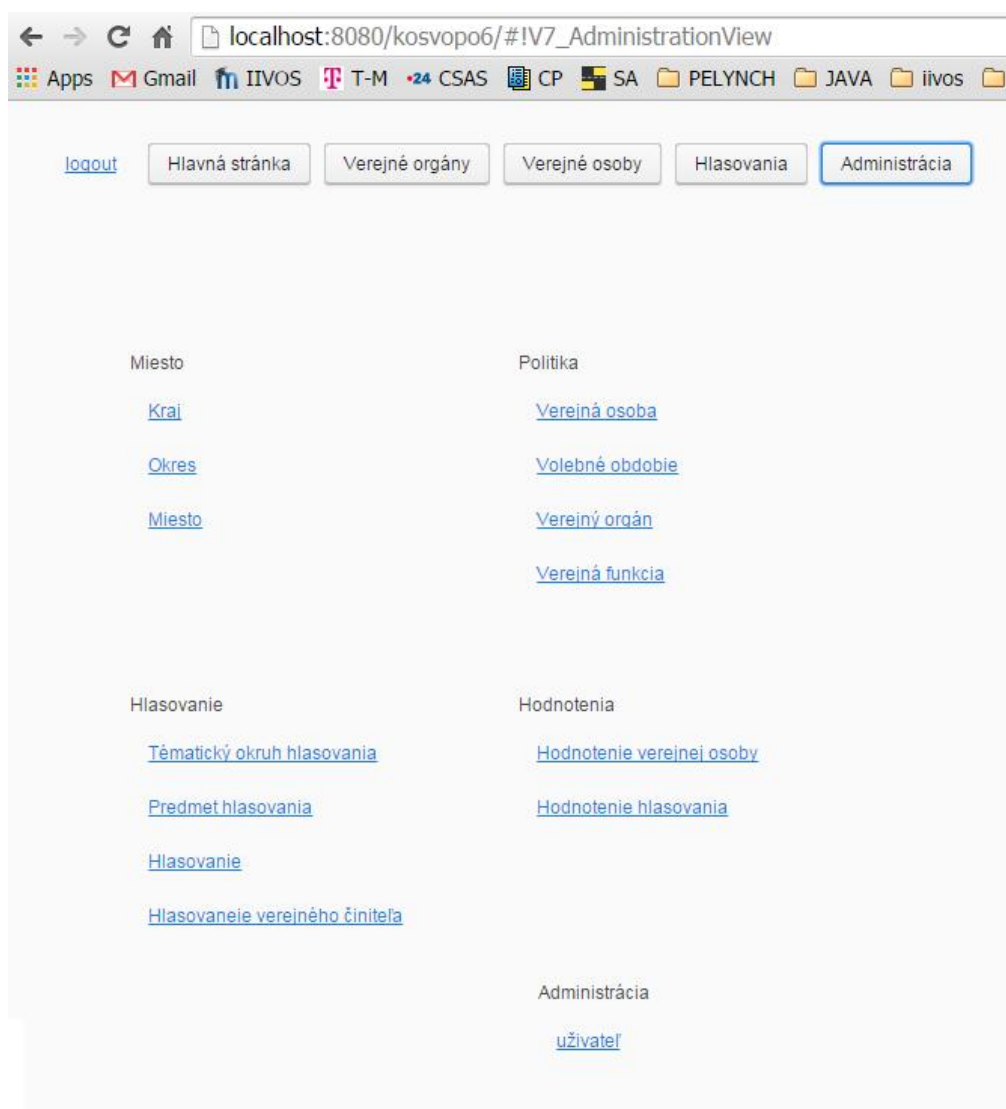


Obr. 19, View 4 – Veřejná osoba – detail, přidávání dokumentů v přihlášení pod administrátorem/dobrovolníkem

Přibude odkaz na stránku (View 7), na které budou odkazy na univerzální správcovské rozhraní pro všechny entity.



Obr. 20, Hlavní menu, administrace.



Obr. 21, View 7 - Administrace.

Samotná editační stránka (View 8) bude univerzální, tj. přizpůsobena všem typům entit. Umožní jak editaci, tak přidávání i mazání entit tříd týkajících se modelu a editace vlastního profilu (hesla a pod).

The screenshot shows a web application interface for managing 'public_person' entities. The browser address bar indicates the URL is `localhost:8080/kosvopo6/#!public_person`. The page has a navigation bar with links: [logout](#), [Hlavná stránka](#), [Verejné orgány](#), [Verejné osoby](#), [Hlasovania](#), and [Administrácia](#).

tabulka se entitami

| id | meno | priezvisko | dátum narodenia |
|----|-------|------------|-----------------|
| 1 | Johan | ProcházkaN | 11-09-1956 |
| 2 | Karol | Čútora | 13-05-1966 |
| 3 | Luboš | Jozefíny | 21-12-1954 |
| 4 | Alex | Spevák | 01-09-1979 |
| 5 | Emil | Čútora | 01-04-1967 |

editační formulář:

meno:
priezvisko:
dátum narodenia:

tlačítka pro přidávání / odstraňování entit:

možeš použiť vyhľadavanie

filtrování entit zobrazených v tabulce / výběru entit pro editační formulář

☐ Miesto ☐ Verejný orgán ☐ Verejná osoba
Bratislava-Rača Mestské zastupiteľs Johan ProcházkaN

Obr. 22, View 8 – detail administrační tabulky.

Obr. 23, detail formuláře pro zadávání nového hlasování.

Mapa stránek administrátora.

Všechno jak pro dobrovolníka. Editace i administračních tabulek, přidávání a mazání nových uživatelů (rozšířené možnosti části „Administrace“). V budoucnosti i přístup k možnosti navrácení systému do stavu v minulosti, resp. umožňující oživování deaktivovaných entit.

Možnosti rozšiřování systému.

Grafická podoba stránek.

Jak již bylo zmíněno v úvodu, půjde o jednu z nejdůležitějších stránek celé aplikace. Majíce na vědomí obtížnost řešení tohoto problému (tj. návrhu *user friendly* grafického zobrazení) zatím jsem se mu detailně nevěnoval a nechávám si na to čas, až budu oproštěn od technických problémů. To již bude mimo rozsah této práce.

Návrat do bodu v minulosti

Díky struktuře ukládání změn bude návrat do minulosti(*stav M*) jednoduchou iterací skrz tabulku *A_Change* od současného dne(*stav S*) až do okamžiku ke kterému se chceme vrátit, přičemž v každém 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 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 triggeru na databázové vrstvě. Tento stav bude potřeba reflektovat i u metod ukládání do databáze, tj. ve smyslu: „do simple step only“ a „do complex step“.

Všechny tyto problémy vyplývají z toho, že náš navrhovaný systém ukládání změn umožňuje jen jednu „větev“ 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 se musel systém značně zkomplikovat, protože by musely 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 A_CHANGE vymazat. Aby poslední řádek v této tabulce odpovídal 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 superadministrátor s přístupem do databáze. Uvedu jako možnost budoucího vývoje, že tyto změny by pak mohly být obhospodařovány v další tabulce, do které by sa zapisovaly změny v tabulce A_CHANGE.

Z důvodu, že se id uživatele ukládá do tabulky změn, musí být i tato a od ní závislé administrátorské tabulky deaktivovatelné (entity se nemohou mazat). 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 A_CHANGE zapisovat budou.

Problém vyššího počtu dobrovolnických skupin.

V případě, že se systém rozšíří, bude třeba odseparovat různé dobrovolnické skupiny, aby si „nechodili do zelí“. Řekněme, že budou 2 dobrovolnicke skupiny, monitorující chování městských zastupitelství v Košicích a Bratislavě. Jak by třeba vypadal 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 veřejný 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-admin 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 různých 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 administrátora (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 sloupec „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 snížení 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 umístění

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. Testování

Až bude aplikace dokončená, její testování provedou členové neziskové organizace na kontrolu činnosti veřejných činitelů v Bratislavě.

H. Zajištění kvality

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

I. Závěr

V zadání absolventské práce je uvedeno, že cílem této práce mělo být vytvořit aplikaci na monitorování činnosti veřejných činitelů (poslanců). Jak jsem uvedl v úvodu, podařilo se mi splnit jenom základní funkční kostru, bez rozpracování grafické podoby, která bude též uživatelsky důležitá. Nepovažuji tuto práci za zdaleka ukončenou. Některé části, kterými se budu zabírat, jsem již výše nastínil a mohé další na mne čekají v záloze. Původně jsem chtěl aby tato práce byla jakýmsi návodem, jak pracovat s Vaadinem, posléze však jsem se zaměřil spíše na logiku fungování a opis použití frameworku Vaadin jsem nerozvíjel, nebo se ho dotkl jen okrajově.

Pokusím se pospat své subjektivní zkušenosti, ke kterým jsem prací na díle došel.

Příště, pokud budu vědět, že budu implementovat složitější logiku, bych se snažil vyhnout použití složitějších vestavěných datových kontejnerů (např. `SQLContainer`), které jsou velice užitečné jen v případě, když požadavky uživatele jsou v mezích toho co kontejner přirozeně poskytuje (tj. základní operace: změna dat, resp vkládání nových / mazání existujících). Jejich nevýhodou je, že jsou neflexibilní pokud jde o rozšíření jejich funkcionality. V našem případě pro dosažení ukládání změn jsem byl nucen dodatečnou funkcionalitu implementovat pomocí *základního JDBC*, který naopak velice flexibilní je. Výsledkem je jakýsi konglomerát ve kterém se dvě různé nezávislé systémy starají o tu samou věc – komunikace s databází. Pokud však chceme v aplikaci využít předdefinované komplikované komponenty (*table* a hlavně *timeline*) nemůžeme se použitím *data-containeru* vyhnout. Částečně je pak možné se výše zmíněnému problému (konglomerát) vyhnout tak, že se *SQLContainer-u* ponechají pouze neinvazivní databázové operace (tj. čtení z databáze) a všechny zápis do databáze přenechat flexibilnějšímu *základnímu JDBC*. Tato situace u mě vznikla i z důvodu, že kromě výsledků počáteční analýzy – jaká technologie je pro projekt nejvhodnější - jsem pociťoval také potřebu obeznámit se s některými prvky Vaadinu a v tomto případě tato potřeba s výsledky analýzy kolidovala. Nevím, jestli za pozitivní stránku věci můžu považovat objevení přírůstku do rodiny údajně vyčerpávajícího seznamu 23 *design patterns* a to konkrétně nového návrhového vzoru "*slepenec*".

Osvědčilo se mi používání *základního JDBC* přístupu do DB, kvůli její vysoké flexibilitě. Cenou za něj je obsáhlejší, méně přenositelný kód.

Dobré zkušenosti jsem měl též s použitím reflexe. Na často spomínané potíže při ladění, jsem prakticky nenarazil a vlastně jediným problémem, který mě v této souvislosti trápí je nižší čitelnost kódu pro toho, kdo kód eventuálně převezme. Radikální zhuštění kódu však nakonec určitě poteší oko každého návrháře. Pro nasledovníka bych měl snad doporučení, že zde více než kde jinde je třeba postupovat vpřed po velice malých krocích (tj. implementace – testování – implementace - ...).

Největší slabinu Frameworku Vaadin vidím v těžkopádnosti s jakou se v něm manipuluje s grafickou stránkou věci, tj. polohou objektů na stránce. Taky to však může být tím, že jsem do grafického použití Vaadinu zatím hlouběji nepronikl. Kdybych se např. teď měl rozhodnout pro tvorbu webové aplikace, asi bych zvolil kombinaci HTML, javascriptu a pro komplikovanější funkcionalitu Javu, při AJAX-ové komunikaci Java vs. Javascript. Jednoduše již proto, že se jedná o jazyky nižší úrovně, se kterou mám celkově dobré zkušenosti.

Zdá se mi, že v co jsem původně doufal mě trochu zklamalo, naopak naopak, z toho, co jsem bral jako nevyhnutné zlo se vyklubal dobrý pomocník. Chtěl bych proto závěrem trochu zpochybnit význam počáteční řešerže literatury, v nasledovních mezích: Člověku, který čte článek o nějaké technologii se může subjektivně zdát, že se rozhodl jen a výlučně na základě přečtených informací a nemusí si být vědom faktu, že hlavní roli při rozhodování sehráli jeho osobní zkušenosti z oboru. Kdo tento kontext předchozích zkušeností nemá, může ze stejné objektivní informace (tj. článku, textu, ...) vyvodit úplně jiný závěr. Zdá se mi, že význam řešerže je silně podmíněn rozsahem zkušeností člověka, který ji jako nástroj rozhodování využívá. V počátečním stavu bez zkušeností je její význam malý (náhodní výběr by vyšel možná na stejno), postupně však jak přibývají uživateli zkušenosti nabývá tento nástroj na síle. Podobný problém je známý i z oboru kognitivních věd - poslech „přehltnutých“ hlásek. Domorodec, který jazykem odmalička mluví je „slyší“ i když tam objektivně nejsou. Cizinec, který se jazyk teprve učí z nich „neslyší“ nic – protože tam skutečně nejsou.

Laskavý čtenář si zajisté povšiml, že některé kapitoly byly napsané veselejší formou. Věřím, že se při čtení nenudil a tato mu čtení mu spíš ulehčila, než znesnadnila. Pokud byl pozorný, jistě zaregistroval i mou snahu zabránit tomu, aby forma byla na újmu obsahu. Děkuji mu za pozornost.

J. Epilóg

Odvrhoľ som všetky čary,
frameworky i manuály,
vládnem už len vlastnou,
slabou silou...

Rozhodnite, kde dokončím svoje žitie.

Tu, v sieni slávy,
či v prepadlisku dejín?

Titul aj so slávou
je už skoro môj, a čo už tam
po krivdách, zrade, tie odpúšťam...

Len opustiť pomôžte mi
pustý ostrov tejto scény!

A do plachiet kiež mi vanie vietor –
vaše zatlieskanie,
a dosiahol som svoj cieľ!

Bez čiar a bez kúziel som však ľahká korisť pre zbesilosť pekla,
nuž o milosť Božiu proste pre mňa,
lebo chybám
v živote sa nevyhýbam.

Slobodu ak získam, vďaka! :

Aj vás nech potom spása čaká! [7]

A moje knihy? (podobne jak viď. vyššie manuály)

Tie navždy pochovám,
hlbšie, než kam olovnica dosiahne!
Ibaže by...

Seznam Literatúry

- [1] Apache Maven, Wikipedie http://cs.wikipedia.org/wiki/Apache_Maven
- [2] Reflexe, Wikipedie [http://cs.wikipedia.org/wiki/Reflexe_\(programov%C3%A1n%C3%AD\)](http://cs.wikipedia.org/wiki/Reflexe_(programov%C3%A1n%C3%AD))
- [3] Spring Framework, Wikipedie http://cs.wikipedia.org/wiki/Spring_Framework
- [4] Marko Gronroos: Book of Vaadin, Vaadin 7 Edition – 1st Revision, Vaadin Ltd, 2013
<https://vaadin.com/download/book-of-vaadin/vaadin-7/pdf/book-of-vaadin.pdf>
- [5] Instalation of tomcat, <https://www.digitalocean.com/community/tutorials/how-to-install-apache-tomcat-7-on-ubuntu-14-04-via-apt-get>
- [6] Deploymnet application on tomcat via manager GUI,
<https://www.youtube.com/watch?v=9X9DA8oVodk>
- [7] W.Shakespeare: The Tempest, ODEON, Translation by Ľubomír Feldek, 2008.
volná interpretace.

Přílohy

Uživatelský manuál.

A. Instalace

Předpokládám, že operačním systémem serveru bude Linux, nebo Solaris. Na serveru, kde bude aplikace v provozu, je potřeba nainstalovat webový server Apache Tomcat (nebo jiný web server) a databázový MySQL server.

1. Instalace MySQL -serveru a taky je dobré mít nainstalovaného mysql-client, pro administrátorské účely:

```
sudo apt-get install mysql-client mysql-client-5.5 mysql-common mysql-server-5.5
```

Pak po nastavení MySQL serveru (přes mysql_secure_installation), vytvoření uživatelských rolí

```
mysql> grant all on *.* to admin@'%' identified by 'admin' with grant option;
```

2. Vytvoří uživatele: např. admin s heslem: admin.

3. Vytvoření databáze.

```
mysql> create database kosvopo5;
```

4. Spuštění MySQL serveru jako služby.

```
sudo service mysql start
```

5. Instalace Tomcatu.

```
sudo apt-get install tomcat7
```

pak, po příslušné konfiguraci tomcatu (viz. [5]) a spustit Tomcat jako službu:

```
sudo service tomcat7 start
```

B. Příprava na straně aplikace.

6. Nastavení properties files. Potřebné nastavení přístupu do databáze se konfiguruje v souboru *dbConn.properties*. Tyto údaje musí sedět s nastavením MySQL serveru viz výše. Tento soubor se v projektu nachází v adresáři *resources*, který se po kompilaci přesune do adresáře WEB-INF/classes.

7. Inicializace databáze. Použít script na vytvoření databázového schématu:

```
root$ mysql -u admin -padmin kosvopo5 < ~/...filepath.../kosvopo5_create.sql
```

případně pak inicializace databáze počátečními hodnotami:

```
root$ mysql -u admin -padmin kosvopo5 < ~/...filepath.../kosvopo5_init.sql
```

8. Kompilace war souboru. V hlavním adresáři projektu spustit:

```
mvn clean install
```

C. Spouštění aplikace

Za předpokladu úspěšného provedení kroků části A a B můžeme přistoupit k instalaci (*deployment*) war souboru na Tomcat server. To je možno dvěma způsoby. Tomcat server, pokud je v chodu, zastavíme.

9a. Mechanické zkopírování souboru do adresářové struktury Tomcatu. War soubor (*kosvopo.war*) zkopírujeme do adresáře *webapps* v adresářové struktúře Tomcatu.

9b. Přidání war souboru pomocí tomcat manager GUI. Viz. Návod [6].

10. Restartujeme *Tomcat* server. Pokud proběhlo vše jak mělo, ve webovém prohlížeči by se po zadání *localhost/kosvopo* (na serveru) resp. *IP.adresa.serveru/kosvopo* (z jiných počítačů) měla objevit vstupní stránka (*View 2*).

D. TroubleShooting

Všechny záznamy z chodu aplikace se budou ukládat do logovacích souborů uložených v adresáři *logs* adresářové struktúře *Tomcatu*. Z nich pak bude moci správce vyčíst, co se v aplikaci / na *Tomcatu* nepovedlo. Po opravě kódu se znovu zkompile *war* soubor a instaluje viz. výše na server.