Bakalářská práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Aspektově orientovaný vývoj adaptivní struktury aplikace pro Java EE aplikace

Nikita Mishchenko
Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Květen 2016
Vedoucí práce: Ing. Jiří Šebek

Poděkování

Chtěl bych poděkovat panu Ing. Jiřímu Šebkovi za vedení a cenné rady při vytváření této bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 24. 05. 2016

....

Abstrakt / Abstract

Bakalářská práce se zabývá úlohou vytvoření Java EE frameworku s použitím aspektově orientovaného přístupu, který umožní efektivně generovat strukturu aplikace pro uživatele v závislosti na jeho chování, a umožnit uživatelům používat aplikace nejvhodnějším způsobem. Při řešení úlohy byl použit programovací jazyk Java verzi 8, Spring framework, IntelliJ IDEA IDE na operačním systému OS X El Capitan. Bylo vytvořeno řešení, které umožňuje vývojářům nakonfigurovat tento framework a použít ho pro různé struktury. Toto řešení neznečišťuje kód a nevyžaduje od vývojáře úsilí navíc.

Klíčová slova: aspektově orientovaný přístup, adaptivní struktura, Java, Spring framework, snížení úsilí pro zlepšení pohodlnosti aplikace pro uživatele.

This bachelor thesis deals with the task of creating Java EE framework using aspect-oriented approach, which will allow effectively generate the structure of the application for the user depending on his behavior, and allow users to use the application most appropriate way. Through the solving of the given task the programming language Java version 8 and Spring framework were used with IntelliJ IDEA IDE on OS X El Capitan. Solution that allows developers to configure and use this framework for different structures was created. This solution does not pollute the code and does not require from the developer any unnecessary effort.

Keywords: aspect-oriented approach, the adaptive structure, Java, Spring framework, reduction of efforts to improve convenience for the user.

Obsah

1 Úvod 1
1.1 Motivace 1
1.2 Cíle práce 1
1.3 Definice 2
2 Rešerše 3
2.1 Kontext 3
2.2 Aspektově orientované programování 4
2.3 Spring framework a Spring AOP 5
2.4 Adaptivní uživatelské rozhraní a Adaptivní struktura aplikace 5
3 Definice problémů 6
3.1 Problém 1: Chování meta-modelu po přidání nových prvků, nebo změně logického smyslu již existujících 6
3.2 Problém 2: Frekvence aktualizace meta-modelu 8
3.3 Problém 3: Podle čeho generovat meta-model 9
3.4 Problém 4: Seskupování prvků do rodiče 10
3.5 Problém 5: Příliš mnoho prvků se stejným rodičem 12
3.6 Problém 6: Příliš hluboký strom 13
3.7 Problém 7: Pojmenování rodiče 14
3.8 Problém 8: Vytvoření meta-modelu pro nové uživatele 15
4 Implementace 16
4.1 Úvod do struktury projektu 16
4.2 Definice použitých technologií, frameworku a termínu 17
4.3 Implementace frameworku 18
4.3.1 Třídy pro definice uživatelských struktur a jejich prvků 18
4.3.2 Třídy pro ukládání uživatelských dát 20
4.3.3 Meta model a třídy pro vytvoření meta modelu 21
4.3.4 Servisy a utility 24
4.4 Požadavky pro použití frameworku 30
5 Testování 31
6 Instalace 34
7 Závěr 35
Literatura 36
A Zadání práce 39
B Zkratky 41
C Obsah CD 42

Tabulky

2.1. Asociace kontextových parametrů do tříd 4
5.1. Cesta do prvků se základní data 32
5.2. Cesta do prvků s zvýrazněním třech prvků 33

Obrázky

3.1. Výpočet aktuální váhy 6
3.2. Výpočet přechodné váhy 7
3.3. Výpočet koeficientu 7
3.4. Úrovni navigační struktury 11
3.5. Mnoho prvků se stejným rodičem 12
3.6. Pozice v sekvenci 12
3.7. Příliš hluboký strom 13
3.8. Generování názvu rodiče 14
4.1. Struktura projektu 16
4.2. Třídy pro definice uživatelských struktur a jejich prvků 18
4.3. Třídy pro ukládání uživatelských dát 20
4.4. Třídy pro vytvoření meta-modelu 22
4.5. Přidání prvků do seznamu který dosáhl svůj limit 23
4.6. Přidání prvků do podseznamu který dosáhl svůj limit 23
4.7. Servisy a utility 24
4.8. Sekvenční diagram vyvolání uživatelem jednotlivého endpointu 25
4.9. Ověření nutnosti generování nových meta-modelu 25
4.10. Kontrola změn a nutnosti generování meta-modelu 27
4.11. Ukládání dát a generování nového meta-modelu v NavigationService servise 28
4.12. Struktura databáze 30
5.1. Výsledek testování v Gatling 31

Kapitola 1
Úvod

1.1 Motivace

Každá aplikace musí poskytovat nějaké uživatelské rozhraní pro komunikaci s uživatelem. Toto rozhraní nemusí umožnit jenom komunikaci, ale musí být pro užíváte co nejvíc přívětivě.

V současné době každý člověk denně používá spoustu různých zařízení, jako například počítač, tablet, mobil. Každé z nich je specifické a má své vlastnosti. Kromě toho, každé zařízení se používá v různých situacích. Velký rozdíl mezi tím, když uživatel sedí vklidu před počítačem, a když spěchá a potřebuje něco rychle ověřit nebo najít a bude tyto úkony provádět z mobilu, kde je určitě menší pracovní plocha a pomalejší připojení na internet. Navíc každý uživatel používá aplikace pro své osobní účely a podle svých schopnosti. Aplikace musí ne jenom zobrazovat konkrétní informace, ale také umožnit co nejrychlejším způsobem najít to, co uživatel potřebuje. Z toho vychází problém, jak nabídnout užívateli co nejlepší řešení konkrétně pro něj a jeho potřeby.

Pro takové aplikace, které závisí na kontextu, je několik řešení. Jedno z nich je naimplementovat rozhraní pro každé zařízení, vlastnosti obrazovky, potřeby uživatele atd. Ale takových kombinací je spousta. Taková aplikace už nebude flexibilní. A toto řešení je velmi finančně a časově náročné. Co se stane, když potřebujeme po nějaké době něco změnit nebo přidat? Budeme muset zkomponovat změny všude a bude to stát dost časových a finančních prostředků.

Proto nejlepším řešením je umožnit aplikace adaptovat se pro konkrétního uživatele a jeho potřeby v závislosti na jeho chování a kontextu.

1.2 Cíle práce

Cílem práce je najít a zadefinovat problémy, na které můžeme narazit při adaptace uživatelského rozhraní pro konkrétního uživatele v závislosti na kontextu. Najít a zadefinovat řešení těchto problém, a zanalyzovat výhody a nevýhody konkrétních řešení, kam by jsme mohli směřovat dál.

Po rešerši dané problematiky je úkolem zadefinovat meta model, který bude obsahovat všechny potřebné atributy pro správné rozhodnutí při adaptaci uživatelského rozhraní.

Dalším krokem bude návrh a implementace aplikačního frameworku pro Java Enterprice Edition, který umožní analyzovat a generovat meta model uživatelského rozhraní v závislosti na kontextu, potřeb a chování uživatele. Implementace bude využívat paradigmatu aspektově orientovaného přístupu. A nakonec, bude vytvořena jednoduchá Java EE aplikace, ve které bude použít a otestován tento framework.

1.3 Definice

V této části jsou zadefinovány hlavní pojmy, které budu velmi často používat v textu této práce. Více specifické pojmy se budou definovat v dalších částech této práce, v závislosti od našich potřeb.

Kontext jsou veškeré informace, které mohou být použity na charakterizaci situace určité entitě. Entitou je osoba, místo nebo objekt, který je považován za relevantní pro interakci mezi uživatelem a aplikací, včetně uživatele a aplikace samotných [1].

Aspektově orientované programování (AOP) je programovací paradigma, která má za cíl zvýšit modularitu programu tím, že umožňuje oddělení části logiky programy do aspektů(cross-cutting concerns). Činí tak tím, že přidá další chování na již existujci kód bez úpravy samotného kódu.

Meta-modelem budu jmenovat stromovou strukturu, která bude generována frameworkem a že které vývojář bude schopny zobrazit uživatelské rozhraní.

Kapitola 2
Rešerše

2.1 Kontext

Pojem kontext je velmi široce používány termín. Existuje spousta různých definice tohoto pojmu, ale chtěl bych vybrat jenom dvě, které nejvíce vyhovují pro tuto práce. První je od Day a Abowd [1] [2]. Definovali kontext, kontextově povědomí a kontextově vědomí aplikace jako:

„Kontext je veškeré informace, které mohou být použity na charakterizace situaci určitě entity. Entitou je osoba, místo nebo objekt, který je považován za relevantní pro interakci mezi uživatelem a aplikací, včetně uživatele a aplikace samotných. Systém je závislý na kontextu (context aware), pokud používá kontext k poskytování příslušné informace a / nebo služby pro uživatele, kde relevantnost závisí na úloze uživatele. Povědomí kontextu (context awareness) je možnost vytvořit kontext. Kontextově závislé aplikace přizpůsobuji podle místa použití, kolekce blízkých lidí, hostitelé a přístupnými prostředky, a jejich změny v čase. Aplikace zkoumá výpočetní prostředí a reaguje na změny.“

A druhá je od Chen a Kotz [3] [2], definovali kontext tím, že rozlišovali, co je relevantní a co je kritické:

„Kontext je soubor stavu v prostředí a nastavení, které buď určuje chování aplikace, nebo ve kterých dojde k události aplikace a je zajímavé pro uživatele.“

Aplikaci které využívají informace z kontextu se jmenují kontextově povědomí. Schmidt definuje to tak [4]:

„Kontextově povědomi je znalost o stavu IT zařízení uživatele, včetně prostředí, situaci, a v menší míře, umístění.“

Kontextově povědomi aplikace využívá informace z kontextu, a naším cílem vymyslet algoritmus který pomůže co nejefektivnějším způsobem tuto informace zanalyzovat a adaptovat strukturu aplikace pro konkrétního uživatele.

Povědomí o kontextu jde rozdělit na aktivní a pasivní [3].

Aktivní povědomí o kontextu - aplikace se automaticky přizpůsobí objevenému kontextu, změnou chování aplikace.

Pasivní povědomí o kontextu - aplikace představuje nový nebo aktualizovaný kontext pro zainteresované uživatele, nebo dělá kontext přetrvávající pro uživatele, abych načíst ho později.

Jak už jsme se zmínili, kontext je docela široký termín. Proto má smysl rozdělit kontext na čtyři třídy podle Reena Hanumansetty [2].

Klasifikace se dělí podle typu změn, a mohou být buď statické, nebo dynamické. Rozdíl je v tom, že pokud parametr se bude měnit během relace, tak bude klasifikován

jako dynamický. Pokud ne, tak bude statickým. Do statických parametrů patří platforma, role uživatele, preference atd. Kde platforma je zařízení na kterém se používá aplikace a její vlastnosti. Do preference by mohli patřit třeba nastavení. Dynamické parametry to jsou různý vlastnosti který se budou měnit, jako, například, připojení k internetu, orientace zařízení(nebo velikost pracovní plochy(velikost okna prohlížeče)), aktivita uživatele.

Další úroveň klasifikace jde rozdělit na funkcionální a prezentační. Do funkcionálního patří parametry, který mění funkčnost back-endu. Například, role. V závislosti na role, uživatel má různé funkcionalitu a možnosti. Nebo nastavení aplikace. Do druhé kategorie patří parametry měnící front-end. Což je například platforma.

Tabulka 2.1 zobrazuje příklad rozložení parametrů do tříd.

	Statické	Dynamické
Funkcionální	Role, identita platforma, preference	Připojení k internetu, čas, lokace
Prezentační	Platforma, preference	Osvětlení, aktivita uživatele orientace zařízení

Tabulka 2.1. Asociace kontextových parametrů do tříd

2.2 Aspektově orientované programování

Aspektově-orientovaný přístup (AOP) [5] [6] [7] je programovací paradigma, které řeší oddělení zodpovědnosti (concerns). Tím se rozumí rozdělení počítačového programu na různé části tak, aby se z hlediska funkcionality tyto části co možná nejméně překrývaly.

Na rozdíl od OOP, které pomáhá oddělení zodpovědnosti na základě implementace jednotlivých tříd, je AOP více zaměřené na oddělování zodpovědnosti než OOP. Řeší totiž problematiku takzvaných průřezových zodpovědnosti (cross-cutting concerns).

Průřezové zodpovědnosti jsou takové zodpovědnosti, které se nedají jednoduše zapouzdřit a vyskytují se na více místech programu. Jedná se například o řízení transakcí nebo o logování průběhu programu. AOP se snaží řešit zapouzdření těchto průřezových zodpovědnosti a zároveň rozdělit program na jednotlivé moduly takzvané aspekty.

Klíčovým konceptem AOP, který řeší rozdělení zodpovědnosti v programu je Join point model (JPM). Tento model umožňuje definovat dynamickou strukturu průřezových zodpovědnosti nezávislé na objektové orientované hierarchii. JPM definuje několik důležitých pojmu.

Join point je místo v programu, kam je možné aplikovat průřezovou zodpovědnost pomocí AOP. Může sem patřit volání metod, inicializace tříd, struktura tříd (jméno třídy, názvy a datové typy atributů a jejich anotace) apod.

Advice je rozšiřující kód, který chceme použít k rozšíření stávajícího modelu. Advice můžeme použít v Join pointech.

Pointcut jedná se o místo v aplikaci, kde je potřeba aplikovat průřezovou zodpovědnost. Jedná se o výběr Join pointu, pro které je aplikován Advice.

Aspect je kombinací Advice a Pointcut, určuje tedy logiku, která je do aplikace vložena a místo, kde bude spuštěna.

Weaving je proces integrace aspektů do specifických míst aplikace tak, že spojí aspekty a hierarchickou strukturu aplikace.

Velká část programovacích jazyků implementuje, nebo využívá knihovny pro aspektove orientované programování.

2.3 Spring framework a Spring AOP

Spring framework je Java platforma, která poskytuje komplexní podporu infrastruktury pro vývoj Java aplikací [8]. Umožňuje vytvoření Jáva EE aplikaci jednodušší. Existuje moduly a rozšíření pro různé účely.

AOP přístup se používá ve Spring frameworku pro poskytování deklarativnych podnikových servisu, a to zejména jako náhrada za EJB deklarativní servisy. Nejdůležitější taková servisa je deklarativní řízení transakcí a umožňuje uživateli implementovat vlastní aspekty [9].

Spring AOP je implementováno v čisté Javě. Není třeba žádná speciální kompilace. Ve Spring AOP není třeba kontrolovat class loader hierarchie, a je tedy vhodný pro použití v servlet kontejnerů nebo v aplikačním serverů [9].

Spring AOP přístup do AOP se liší od většiny jiných AOP frameworku. Cílem není poskytnout nejvíc kompletní AOP implementace (i když Spring AOP je docela schopný); je to spíše poskytnout úzké integrace mezi AOP implementace a Spring IoC(Inverse of control), aby pomohla řešit společné problémy v podnikových aplikacích [9].

2.4 Adaptivní uživatelské rozhraní a Adaptivní struktura aplikace

Adaptivní uživatelské rozhraní (také známo jako AUI) je druh uživatelského rozhraní, které se dokáže adaptovat dle profilu uživatele a kontextu. Tato kontextově závislá uživatelská rozhraní lze rozdělit na adaptivní, adaptabilní a kombinovaná. Adaptivní uživatelské rozhraní je takové, ve kterém systém aktivuje kontextové změny. Adaptabilní uživatelské rozhraní je takové, ve kterém uživatel iniciuje změnu manuálně. Kombinované predstavuje kombinaci obou přístupu [7].

Adaptivní Struktura Aplikace (Adaptive Application Structure (AAS)) je taková, která mění svou strukturu na základě aktuálního kontextu. To přináší výhody pro koncové uživatele, protože každý má své vlastní potřeby. Aplikace je vytvořena nějakým způsobem vývojáři, ale každý uživatel se používá aplikace různém způsobem.

Kapitola 3
Definice problémů

3.1 Problém 1: Chování meta-modelu po přidání nových prvků, nebo změně logického smyslu již existujících

Jedním z nejdůležitějších faktorů hodnocení prvků je váha jednotlivých prvků. **Skutečnou váhou prvků** budeme nazývat frekvenci použití konkrétního prvků uživatelem. Meta-model musí mít atribut do kterého se bude přepočítávat aktuální počet využití prvků uživatelem.

Jakákoliv dobrá aplikace se časem postupně se mění a rozšiřuje se. Přidávají se nové funkci, mění se již existující nebo odstraňují se něco co už není potřeba. Tyto změny určitě musí ovlivňovat meta-model uživatelů které používají tuto aplikace. Z toho dostáváme problém, jak udělat ty změnu co nejvíc pohodlním způsobem pro uživatele a přitom neztratit se nové možnosti aplikace v již zformovaném meta-modelu.

Řešením je přidání dalších atributu. Jeden a nejdůležitější z nich je **přechodná váha prvků**. Je to váha, do které se nebude připočítat počet použití prvků (ten se bude připočítat se ke skutečné váhe), ale bude mít vliv na hodnocení prvků. Z toho dostávám další váhu, která se jmenuje aktuální váha prvků.

Aktuální váha prvků se skládá ze skutečně váhy a z přechodné váhy.

$$Váha_{aktuální} = Váha_{skutečná} + Váha_{přechodná}$$

Obrázek 3.1. Výpočet aktuální váhy

Přechodná váha se jmenuje přechodnou z důvodů, že slouží jenom na to aby nové prvky měly šanci být zviditelněny pro uživatele a neztratily se někde hluboko. Za nějakou dobu musí být tato váha nahrazena nulou.

Každá aplikace je unikátní a slouží pro různé účely. Například, je velký rozdíl mezi nějakou zábavní aplikace, kterou uživatel používá několik krať denně, když má přestávku, nebo večer, když má odpočinek, zpravodajský portál, který uživatel čte třeba odraná z kávou, nebo nějaký informační systém, který může navštěvovat jen několik krať za měsíc. Programátoři které podporují aplikace určitě budou vědět pro jaké účely je tato aplikace, pro koho, a jak často se využívána. Například, existují aplikace, pro administraci webového hostingu, kterou běžný uživatel používá většinou pro prodloužení objednaných služeb nebo pro změnu osobních údajů. Z těchto důvodů vyplývá další možnosti.

Délka existence této váhy může být buď nastavená programátorem, nebo bude mít výchozí hodnotu 14 dnů. Důležitou poznámkou je, že doba existence se bude započítat jen dní, kdy uživatel se používá tuto aplikace. Tak to uděláno z důvodů, že když uživatel

používá aplikace pár krát za měsíc, nebo v konkrétním obdoby odjel někam na dovolenu, atd., a neměl možnost nebo nutnost využívat aplikace, ty změny prostě neproběhli mimo něho.

Další věc je samotná hodnota přechodné váhy, kterou bude nastavovat programátor. Například, opět z důvodů, že každý zná svůj systém lépe než kdokoli jiný, a z důvodů, že by chtěl nějak zvýraznit nové možnosti aplikace, nebo upozornit uživatele na nějaké změny ve funkcionalitě. Ale okamžitě narážíme na problém, jakou hodnotu musíme zadat této váze, protože každý uživatel používá aplikaci různým způsobem. Třeba nějaká sociální síť má desítky, možná stovky kliknutí od jednoho uživatele denně, ale internetové bankovnictví tolik mít nebude. Jeden uživatel může mít tři nejoblíbenější prvky a mít kolem 700 kliku na každý, druhý uživatel, který se používá tu samu aplikace déle bude mít 2000 kliku na každý prvek. Z toho důvodů musím zavést další atribut a to bude koeficient, který bude pro každého uživatele různé.

$$Váha_{\text{přechodná}} = Váha_{\text{nastavená programátorem}} \times Koeficient$$

Obrázek 3.2. Výpočet přechodné váhy

Koeficient musí mít krok, opět z důvodů různých účelu aplikace. Výchozí hodnota bude 100, ale to taky jde změnit, a tým zjednodušit výpočty. Podle kroku se bude vypočítávat se koeficient.

$$Koeficient = \sum_{\text{všech skutečných vah prvků}} \div Počet\ prvků \div Krok$$

Obrázek 3.3. Výpočet koeficientu

Tím bude vyřešen problém přechodné váhy pro různý uživatele, například, když jeden uživatel bude mít dohromady 1000 kliknutí a 10 prvků, koeficient s výchozím nastavení bude 1, a druhý bude mít dohromady 500 kliknutí a 10 prvků, koeficient bude 0.5. Tak při zadání váhy prvků 50, každý uživatel bude mít různou přechodnou váhu, v závislosti na době použití a intenzitě použití aplikace.

Nevýhody:
Nevýhodou je to, že se docela těžko předpovídá chování každého uživatele ve velké aplikaci a určitě se někde budou vyskytovat malé chyby vyrovnáni vah.

Výhody:
Výhodou je to že tento postup je docela dobře pomůže přidávat/měnit funkcionalitu, přičemž neporušovat logiku a již existující meta-model, a zároveň dávat pozornost uživatele na změny a možnost najít něco užitečného v nových funkcích.

Nové atributy pro meta-model:

- Skutečná váha
- Přechodná váha
- Počet zbývajících dnů do vynulování přechodné váhy
- Koeficient skutečné váhy
- Krok koeficientu přechodné váhy

3.2 Problém 2: Frekvence aktualizace meta-modelu

Každý uživatel během jedné seance komunikace s aplikace obvykle používá několik různých prvků a to znamená, že se mění skutečná váha těchto prvků. Z toho vyplývá, že se musí měnit struktura meta-modelu uživatele. Navíc aktualizovaná data se vždy hodí. Ale pro naší situaci to není úplně správné. Takový postup má několik velkých nevýhod:

- Zbytečně obtěžování výpočetních zdroje systému
- Pokaždé nová struktura

První je to že musíme pokaždé přepočítávat strukturu meta-modelu a to trvá nějaký čas a používá se zdroje systému, což je naprosté není možné pro aplikace s velkým počtem uživatelů. A navíc je to zvyšuje dobu odezvy.

Druhou nevýhodou je to že struktura vždy bude se měnit a uživatel pokaždé bude vidět prvky v různém pořadí. Určitě je v tomto případě a výjimky, třeba v případě, když uživatel používá jenom jeden prvek, ale ve většině případů ta struktura bude prostě skákat. Velku role v uživatelské přívětlivě struktuře hraje zvyk. Ani v případě, kdy aplikace má hrozně ošklivou strukturu, uživatel, který používá tuto aplikace docela dlouho, zvykne se na rozmístění jednotlivých prvků. Proto změny(aktualizace) meta-modelu musíme provádět velmi pečlivě a nedělat to příliš často.

Jak už bylo řečeno, docela mnoho věci závisí od toho pro jaké účely se slouží aplikace, navíc je velmi dobrým nápadem udělat ji co nejvíce pružnou a konfigurovatelnou. Proto další atribut meta-modelu - frekvence aktualizace, bude mít možnost nastavit vývojář, ale samozřejmě bude tento atribut mít výchozí empirickou hodnotu nastavenou na 100. Musíme někam ukládat kdy naposled byla provedena aktualizace meta-modelu, proto musíme zavést atribut, který se bude jmenovat hodnota poslední aktualizace.

Hodnotou poslední aktualizace budu jmenovat součet skutečných vah všech prvků meta-modelu, při kterém naposled byla provedena aktualizace meta-modelu.

Frekvence aktualizace je celkový počet změn skutečně váhy všech prvků meta-modelu, přes který musím provést aktualizace tohoto meta-modelu.

Zadefinoval jsem v jakém případě musím provádět aktualizace meta-modelu, ale je ještě jedna věc, která by zlepšila provedení aktualizace a ušetřila zdroje systému. Když uživatel něco dělá s aplikace, a najednou jemu se změní struktura aplikace, tak první otázkou bude: `co se stalo`. V případě, kdy uživatel někam spěchá, tak ta změna zvyklé struktury bude úplně mimo. A ověřování po každému kliknutí na prvek, jestli nenastal ten samy moment, kdy musíme aktualizovat meta-model, bude nám krást zdroje a čas. Řešením toho problémů bude provedení kontroly při zahájení uživatelského seance (přihlášení uživatele).

Existuje ještě jedna podmínka, při které musí být provedena vynucená aktualizace meta-modelu. Je to změny ve funkcionalitě aplikace (například, přidání nových prvků nebo změna již existujících). Uživatel nesmí vidět nepovolené, nebo už neexistující možnosti aplikace, proto aktualizace musí být provedena v nejbližší možné době. Takže z důvodů že některý prvky který měli nějakou skutečnou váhu mohli být odstranění při změnách, musíme nastavit hodnotu poslední aktualizace na současný součet skutečných vah všech prvků meta-modelu, jinak by jsme mohli čekat na další aktualizace velmi dlouho.

Konečná definice provedení aktualizace meta-modelu bude vypadat takto:

Provedení aktualizace meta-modelu struktury aplikace pro konkrétního uživatele bude proběhat při začátku uživatelského seance (přihlášení uživatele) v případě, když součet

hodnoty poslední aktualizaci a frekvence aktualizace bude menší nebo roveň součtu skutečných vah všech prvků meta-modelu, nebo při změnách ve funkcionalitě aplikace. V případě když, byli provedeny změny ve funkcionalitě aplikace, aktualizace musí být provedena okamžitě a hodnota poslední aktualizaci bude nastavená na součet všech skutečných vah, všech zbývajících prvků meta-modelu.

Nevýhody:

Je nutná nějaká doba na učení, abych se to začalo fungovat. Možná pár prvních aktualizace struktura se bude aktivně měnit, ale tohle závisí na nastavení.

Výhody:

Neobtěžující systém výpočty. Docela přesná aktualizace. Uživatel nemusí po každém svém kliknutí zvykat se na novu strukturu.

Nové atributy pro meta-model:

- Hodnota poslední aktualizace
- Frekvence aktualizace

3.3 Problém 3: Podle čeho generovat meta-model

Meta-model je velká abstrakce, což nám spolu s aspektově orientovaným přístupem umožňuje udělat náš nástroj velmi pružný a univerzální. Meta model může popisovat jakýkoliv typ struktury, buď navigační menu, widgety atd.

Také uživatel má spoustu atributů, které se ovlivňuje ho komunikace s aplikace. A na tyto atributy musíme dávat pozornost. Z toho vyplývají otázky, z čeho musíme formovat meta-model a na čem bude záviset jednotlivé meta-modely. Jinými slovy co bude unikátním identifikátorem meta-modelu.

Existuje mnoho různých faktorů, které ovlivňují vlastnosti meta-modelu: platforma, vzhled, typ a účely aplikace atd.

V současné době, praktické všichni stacionární počítače a notebooky mají docela velký obrazovky, a s nimi nebudeme mít žádný problém. Ale máme ještě různé mobilní zařízení. Povinným atributem je přizpůsobení pro mobilní zařízení. Nejprve je to komfort pro uživatele a v neposledně řadě to že většina systémů a vyhledávače dávají pozornost a prioritu aplikaci, která je adaptována pro různé typy zařízení. Proto musíme mít možnost měnit velikost meta-modelu.

Uživatel vždycky má nějaké role a v některých případech více než jednu, nebo jedna role dědí od jiné/jiných. Například, existuje obyčejný uživatel, který má nějaké základní možnosti jako editace profilu, změna hesla a jiné. Například moderátor má samé možnosti jako normální uživatel ,ale navíc ještě možnost blokovat různé uživatele. Nebo, v případě, kdy každá role odpovídá za konkrétní část aplikace, oprávnění uživatele, se budou skládat z několika roli.

Jak už to bylo označeno, meta-model může popisovat jakýkoliv typ struktury, která se působí komunikace uživatele a aplikace. Aplikace může mít několik takových typu struktur v jedněm pohledu. Pod pohledem v této situace rozumím to co vidí uživatel před sebou, když používá aplikace. Například, je nějaké měnu, pak obsah stránky z několika bločku, nebo v e-shopu typické používá se průvodce založený UI(Wizard based UI) při vytvoření objednávky. Uživatel musí, například, vyplnit dodací údaje, fakturační, způsob platby a doručení atd. Měnu, ve většině případů, je na stránce vždy, a

toto měnu uživatel používá mnohem častěji, a pro jiné účely. Z čeho plyne že je to úplně různé struktury pro různé účely, a bylo by rozumné generovat pro ní různé meta-modely.

Každý jednotlivý prvek musí mít svůj unikátní identifikátor (unikátní jméno), z důvodů že pak z těchto prvků se bude vygenerovaná uživatelská struktura, ze které pak bude vytvořen meta-model. A v případě jakýchkoliv změn by jsme museli kontrolovat co se změnilo a co ne, a neztratili data o chování uživatele.

Že všech těch důvodů, musíme povolit programátorů osobně definovat uživatelské struktury. Programátor musí být schopen řešit co a kam bude patřit. Jeden prvek může patřit do několika různých uživatelských struktur, ale pro každou strukturu bude mít různou statistiku použití. Tak programátor bude schopen řešit co, kam, pro jakou roli nebo zařízení zobrazovat.

Jako výsledek, můžeme říci, že meta-model je možné jednoznačně identifikovat pomocí uživatele a názvu struktury.

Výhody:

Umožňuje vyhnout se duplicit. Dát možnost programátorů řešit co, kde a jak zobrazovat. Velká pružnost.

Nové atributy pro meta-model:

- Unikátní název prvků
- Unikátní název struktury

3.4 Problém 4: Seskupování prvků do rodiče

Aby jsme měli možnost nějak seskupovat prvky, oni musí mít ještě několik atributů kromě již existujících. To nám pomůže jejich nějak logické seskupit.

Nejprve musíme řešit, co je rodičem musí-li rodič být kompletním prvkem, nebo bude sloužit jenom jako logicky oddělovač.

Ve většině případů, nepotřebujeme abychom rodič měl svou funkčnost, z důvodů že to není pohodlné, a málo kde se to používá. Navíc nás to bude obtěžovat při rozhodování u optimalizace struktury. Kdybychom potřebovali přidat nějakou funkcionalitu, tak můžeme přidat další normální prvek. Proto budeme používat rodiče jako logicky separátor. Pomůže nám to tím, že pro efektivnější seskupování můžeme klidně ho odstranit a přidat nový, který bude odpovídat nové skupině.

Logicky význam prvků můžeme rozdělit na čtyři kategorie:

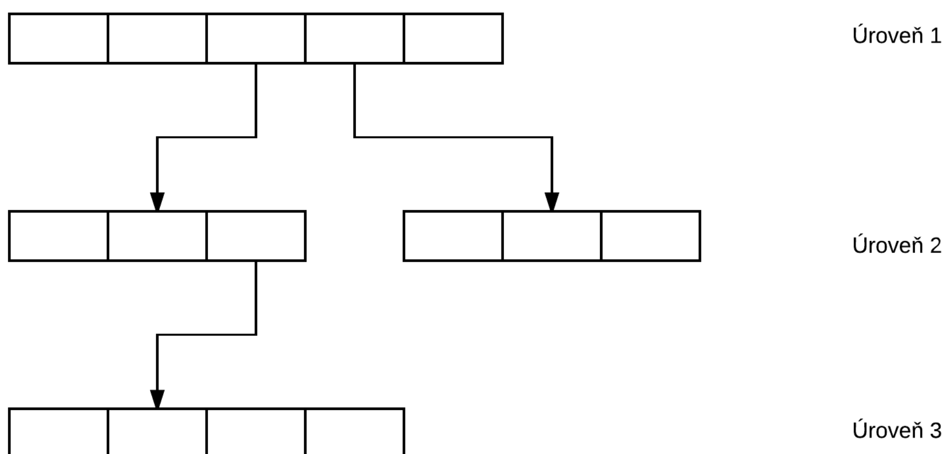
- **Vytvoření** (create) - prvek vede nás na vytvoření něčeho. Například, přidání nového článku, nebo vytvoření uživatele, nebo napsání zprávy atd.
- **Čtení** (read) - prvek vede nás na čtení něčeho. Například, čtení článku nebo aktuálních novinek, nebo zobrazení profilu uživatele atd.
- **Editace** (update) - prvek vede nás na editace něčeho. Například, změna hesla nebo osobních údajů, změně nastavení, opravě komentáře atd.
- **Smazání** (delete) - prvek vede nás na smazání/zrušení něčeho. Například, smazání zprávy, stornování objednávky nebo platby atd.

Dostáváme z toho atribut který budeme používat při pojmenování rodičovských prvků a bude se jmenovat **kategorie akce** může mít hodnoty Vytvoření / Čtení / Editace / Smazání.

Seskupovat prvky musíme nejprve podle jejich aktuální váhy. A musíme udělat to tak, aby cesta do prvků které uživatel používá nejčastěji by byla minimální. Pod cestou rozumím počet rodičů kterých musíme projít aby dostat do cílového prvků. Proto musíme klasifikovat prvky do třech skupin podle priority:

- Důležité - prvky s velkou aktuální vahou, tj. prvky které uživatel používá často. Musíme na nich dávat zvýšenou pozornost a posunovat co nejvíc nahoru.
- Normální - prvky se střední aktuální vahou, tj. prvky které uživatel někde používá. Bylo by dobře posunout jejich co nejvíc nahoru, ale není to hlavním cílem, pokud to z nějakého důvodů nejde tak nechat jak to je.
- Nevadí - prvky s nízkou aktuální vahou, tj. prvky které uživatel praktické nepoužívá. Nemusíme s nimi nic dělat pokud nevzniká nějaký problém zadefinovany v této práce.

K tomu seskupování budeme používat algoritmus K-means. Zvolíme 3 centroidy. Pro nejvyšší prioritu to bude maximální váha, pro nejnižší minimální a pro střední to bude střední hodnota všech vah.



Obrázek 3.4. Úrovni navigační struktury

Kazda uživatelská struktura má několik úrovni [Obrázek 3.4]. A v nejvyšším úrovni (Úroveň 1) musíme zarezervovat několik míst pro prvky s vysokou prioritou. Zbyly místa budeme používat pro seskupení ostatních prvků.

Nevýhody:

Může se nastat to že prvky které patří do jednoho rodiče budou mít málo společných vlastnosti.

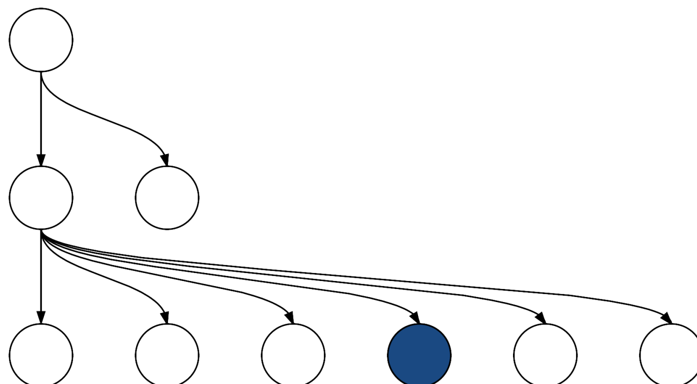
Výhody:

Docela vhodné řešení, které nám pomůže seskupit prvky a při tom nepokazit logiku.

Nové atributy pro meta-model:

- Kategorie akce

3.5 Problém 5: Příliš mnoho prvků se stejným rodičem

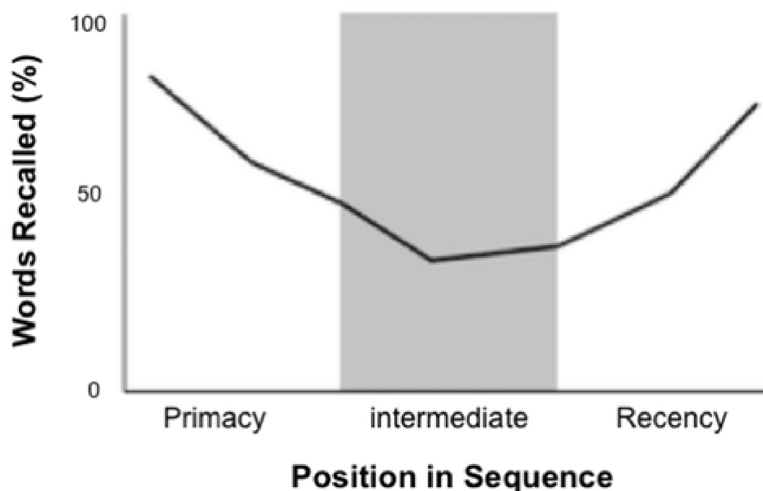


Obrázek 3.5. Mnoho prvků se stejným rodičem

Jeden z nejdůležitějších problémů vzniká když pod jedním rodičem se nacházejí příliš mnoho prvků, je vidět z obrázku 3.5. Člověku, i v případě kdy on používá systém nějakou dobu, bude těžko najít nutný prvek mezi velkého počtu jiných.

Podle Andy Crestodina [10], i osm prvků může být příliš mnoho. A to je z důvodů že krátkodobá pamět člověka drží jenom sedm prvků. To znamená, že osm je mnohem víc než sedm. Z několika prvky, oči uživatele, pravděpodobně, bys mohli najít důležitější prvky. Čím méně prvků, tím každý z nich je výraznější. Tak nevhodnějším počtem prvků je 5. A funguje to pro různé struktury, například, a pro strukturu nějakého pohledu(stránky, vzhled).

Kromě toho, prvky které se objevují jako první nebo poslední v seznamu jsou nejvíce efektivní, a navigace není výjimkou. Výzkum psychologie [10] [11] ukazuje, že pozornost a uchovávání je vyšší pro věci které se objevuje na začátku a na konce. To se jmenuje `efekt sériové pozice` (serial position effect), a je založen na zásadách nadřazenosti a aktuálnosti.

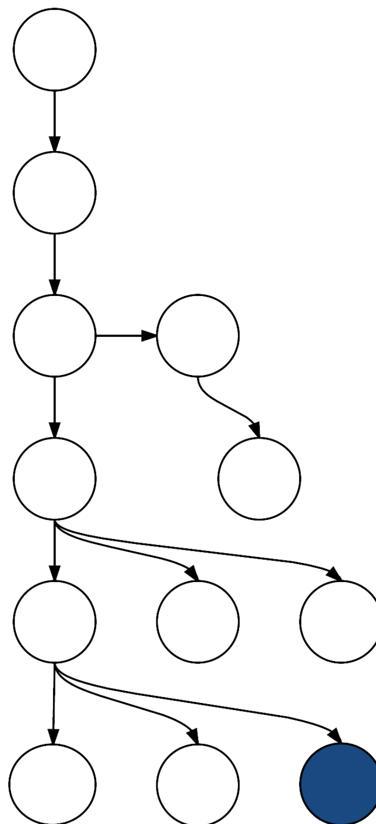


Obrázek 3.6. Pozice v sekvenci (převzato z [10])

Z toho plyne, že kromě toho že musíme mít maximálně 7, a lepší by bylo mít počet prvků pod jedním rodičem méně nebo rovno 5, tak ještě prvky s nejvyšší aktuální vahou v této skupině musí se nacházet na začátku a na konce. A ten zbytek, musíme dát do jiné skupině.

Výhody:
Pohodlná navigace pro uživatele.

3.6 Problém 6: Příliš hluboký strom



Obrázek 3.7. Příliš hluboký strom

Problém vzniká když máme příliš mnoho vnořených prvků, což samozřejmě není dobře. Uživatel bude muset proklikat každý prvek abych dorazit do cíle. Trvá to docela dlouho a dráždí se to. Pro nové uživatele to bude nepřijatelné a člověk může prostě ztratit se a nenajít hledaný prvek. To jsme nemůžeme dovolit.

Řešení toho problémů je faktické popsáno v bodě 3.5. Problémy jsou docela podobně a musíme dávat pozornost jak na počet prvků u jednoho rodiče, tak na hloubku.

Nevýhody:
Nevýhodou je to, že určitě existuje situace kdy těch prvků je opravdu příliš mnoho a nemůžeme jejich nějak seskupit do jednoho stromů tak aby nebyl příliš hluboký nebo příliš „široký“.

3.7 Problém 7: Pojmenování rodiče

Pro pojmenování rodičů musíme vědět jaké prvky každý z nich obsahuje, a jak můžeme této prvky charakterizovat.

Data, které máme v názvu prvků nám nebudou stačit. Potřebujeme to více specifikovat. Některé prvky při seskupení mají málo společného, ale potřebujeme něco s tým vymyslet. Navíc musíme dát svobodu vývojářů nějak ovlivnit té generovaný názvy.

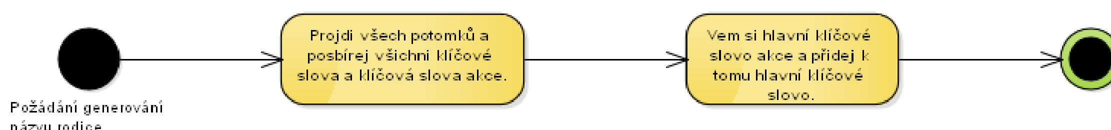
Řešením tohoto problémů bude přidání dalšího atributu, a takovým atributem budou **klíčové slova** nebo tagy.

Z matematického pohledu atribut klíčové slova by byl množinou řádku, popisujících konkrétní prvek pro běžnou strukturu. Tak by jsme mohli najít průnik těchto množin a seřadit podle počtu výskytu jednotlivých klíčových slov. Ten element, který vyskytuje u většího počtu prvků by byl pro tuto situace hlavním. Té klíčová slova by mohli mít hodnoty, například, „Product“, „Items“, „Order“ atd. Abych zvýraznit jedno nebo pár z těch klíčových slov, můžeme jejich tam přidat několik krát, tím zvětšit se počet jejich výskytu.

Ale při pojmenování rodičů nám by nestačila jenom ta skupina klíčových slov. Musíme ještě popsat akce toho co s té prvky můžeme provádět, abych ten název byl více specifikovány. Přidáme tam podobným způsobem další skupinu klíčových slov, která se bude jmenovat akční klíčové slova, a bude fungovat stejně. Ten atribut nebude povinným, a v případě když ho nenastavíme vezme se to z atributů kategorie akce. Atribut akční klíčová slova může mít hodnoty, například, „Add“, „Manage“, „Modify“ atd.

S použitím těch dvou množin klíčových slov by jsme mohli dostat název „Manage products“. A tím dáváme programátorů možnost ovlivnit generování těch názvu. Odhadnout všichni možné varianty je docela problematické, ale člověk který to vyvíjí bude o tom vědět víc.

Algoritmus pojmenování rodiče bude využívat atributy kategorie akce, hlavní klíčové slovo pro tuto skupinu a hlavní akční klíčové slovo pro tuto skupinu. Viz obrázek 3.8.



Obrázek 3.8. Generování názvu rodiče

Pro rodiče, které v sobě budou obsahovat dalších rodičů to bude fungovat podobným způsobem. Ale v takovém případě budeme používat klíčové slova potomkův.

Nevýhody:
Může se nastat situace když pojmenování nebude úplně ideálním.

Výhody:
V pomoci správného zadání klíčových slov jde dostávat pěkně názvy.

Nové atributy pro meta-model:

- Klíčové slova
- Akční klíčové slova

3.8 Problém 8: Vytvoření meta-modelu pro nové uživatele

Určitě každý uživatel bude mít svůj osobní meta-model. Ale ten meta-model bude vytvořen za nějakou dobu po začátku využití aplikace, z důvodů že aplikace se potřebuje naučit se pro každého uživatele, abych odpovídala konkrétně jeho potřebám.

Když máme nového uživatele, aplikace ještě nic neví o jeho chování, ale musí nabídnout nějakou model navigace. Jednou z variant je nabízet nějaké základní meta-model pro všichni uživatele. Proto meta-model pro nové uživatele, nebo pro novu strukturu bude generován z přechodných vah jednotlivých prvků s koeficientem 1.

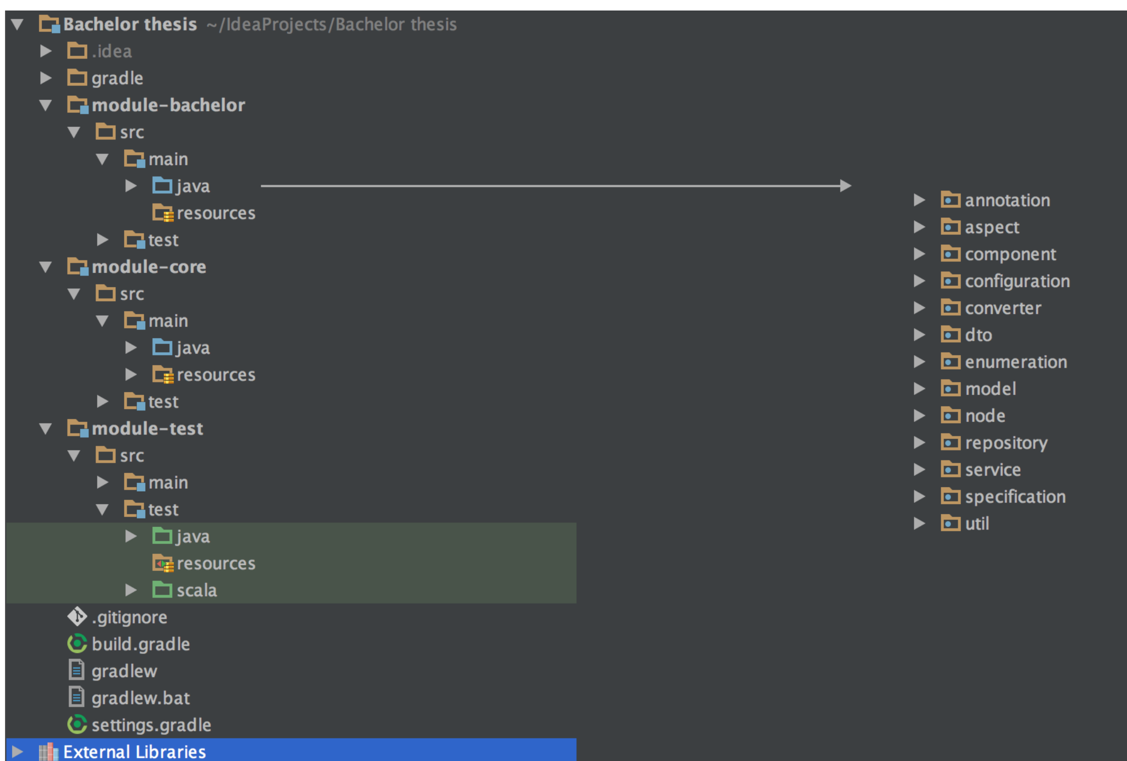
Nevýhody:
Každý uživatel je unikátní a potřebuje svůj osobní meta-model.

Výhody:
Umožňuje docela jednoduché nabídnout uživatele co nejvhodnější rozhraní pro komunikace s aplikace.

Kapitola 4
Implementace

4.1 Úvod do struktury projektu

Implementace se skládá ze třech modulu, jde to vidět na obrázku 4.1.



Obrázek 4.1. Struktura projektu

Module-bachelor obsahuje implementace frameworku a na obrázku ?? můžeme vidět jeho strukturu.

Module-core obsahuje jednoduchou REST aplikace pro testování našeho frameworku. Tato aplikace má frontend který používá REST endpointy pro zobrazení a modifikace dát. Pomoci frontendu můžeme jsme vytvořit nového uživatele, přihlásit se, a proklikat jednotlivé prvky navigační struktury. Frontend komunikuje že komunikuje pomoci technologie AJAX. Navíc tento module obsahuje v sobě veškerou konfigurace. Používá se rozhraní Liquibase pro generování struktury databáze a Spring boot s Jetty serverem pro spuštění aplikace. V této konkrétní aplikace mám nakonfigurované spojení s databasymi Postgres a H2, ale jde jednoduché přidat další. Pro komunikace s databáze používám Hibernate a Spring Data JPA, což nám umožňuje jednoduché manipulovat s entity. Tato aplikace obsahuje logiku pro přihlášení uživatele a používá pro to Srping security. Nic složitého v tom není, každý uživatel má uživatelské jméno a heslo, a slouží pro generování meta-modelu a sběru dát pro konkrétního uživatele.

Module-test slouží k testování chování našeho frameworku a simuluje kliknutí uživatele na element navigační struktury pomoci Scala frameworku Gatling. Máme v něm seznam všech endpointu a ten test zvolí náhodným způsobem jeden z endpointu a zavolá se ho od uživatele.

Buildeni té celé aplikace probíhá pomoci nástroje Grádle.

4.2 Definice použitých technologií, frameworku a termínu

Scala je multiparadigmatický programovací jazyk navržený tak, aby integroval rysy objektově orientovaného a funkcionálního programování [12] [13].

AJAX (Asynchronous JavaScript and XML) je v informatice obecné označení pro technologie vývoje interaktivních webových aplikací, které mění obsah svých stránek bez nutnosti jejich kompletního načítání znovu za pomoci asynchronního zpracování webových stránek pomocí knihovny napsané v JavaScriptu ??.

REST (Representational State Transfer) je architektura rozhraní, navržená pro distribuované prostředí [14] [15]. Umožňuje nám jednoduché komunikovat z aplikace pomoci různých metod HTTP protokolu.

ORM (Object-relational mapping) je programovací technika v softwarovém inženýrství, která zajišťuje automatickou konverzi dát mezi relační databází a objektově orientovaným programovacím jazykem [16].

Hibernate je ORM framework který implementuje JPA (Java Persistence API). Umožňuje vývojářům snadně zachování dát [17].

Spring Data JPA je součástí skupiny Spring Data. Umožňuje snadné implementovat JPA repositary. Tento module zabývá zlepšením podpory pro JPA založené vrstvy pro přístup k datem. Umožňuje jednodušší psát Spring aplikaci které pracují s data [18]. Už obsahuje základní operace s entity a docela jednoduché jde přidat další. Například, pokud máme entitu User a chceme najít uživatele podle uživatelského jména(username) stačí přidat do JPA repositáře metodu která se bude jmenovat findByUsername, a bude mít jeden argument username. Nic víc nemusíme dělat, Spring všechno udělá sám.

Spring boot je další části Springu. Umožňuje nám vytvářet hotové k spuštění aplikace a velmi snadnou jejich konfigurace pomoci `.properties` nebo `.yaml` souboru. Nemusíme bojovat se serverem, Spring boot zabalí server a naší aplikace do takzvaného fat jaru, a zbýde nám jen spustit jar soubor a budeme mít funkční Jáva EE aplikace.

Spring security je silný a vysoce přizpůsobitelné framework pro autentizace a kontrolu přístupu. To je de-facto standard pro bezpečnost Spring aplikace [19].

Jetty server je Javovy webový server.

Gatling je framework napsaný v programovacym jazyce Scala pro testování zatížení (load testing). Umožňuje snadné a rychle volat různé endpointy s různé data velmi rychle. Přičemž jde nastavovat spoustu různých atributů, jako hlavičky požadavků, autentizaci atd.

Gradle je nástroj pro automatizace sestavení aplikace. Je napsán v programovacím jazyce Groovy a je velmi flexibilní.

Liquibase je nezávislá na databáze knihovna pro sledování, správu a uložení změn schématu databáze. Umožňuje nám nakonfigurovat strukturu databáze, a používat pro jakoukoliv data báze, přidávat a sledovat změny.

Postgres databáze je otevřený objektově-relační databázový systém.

H2 databáze je relační databázový systém napsaný v Java. Umožňuje vytvářet databáze v paměti, což je výhodné pro testování.

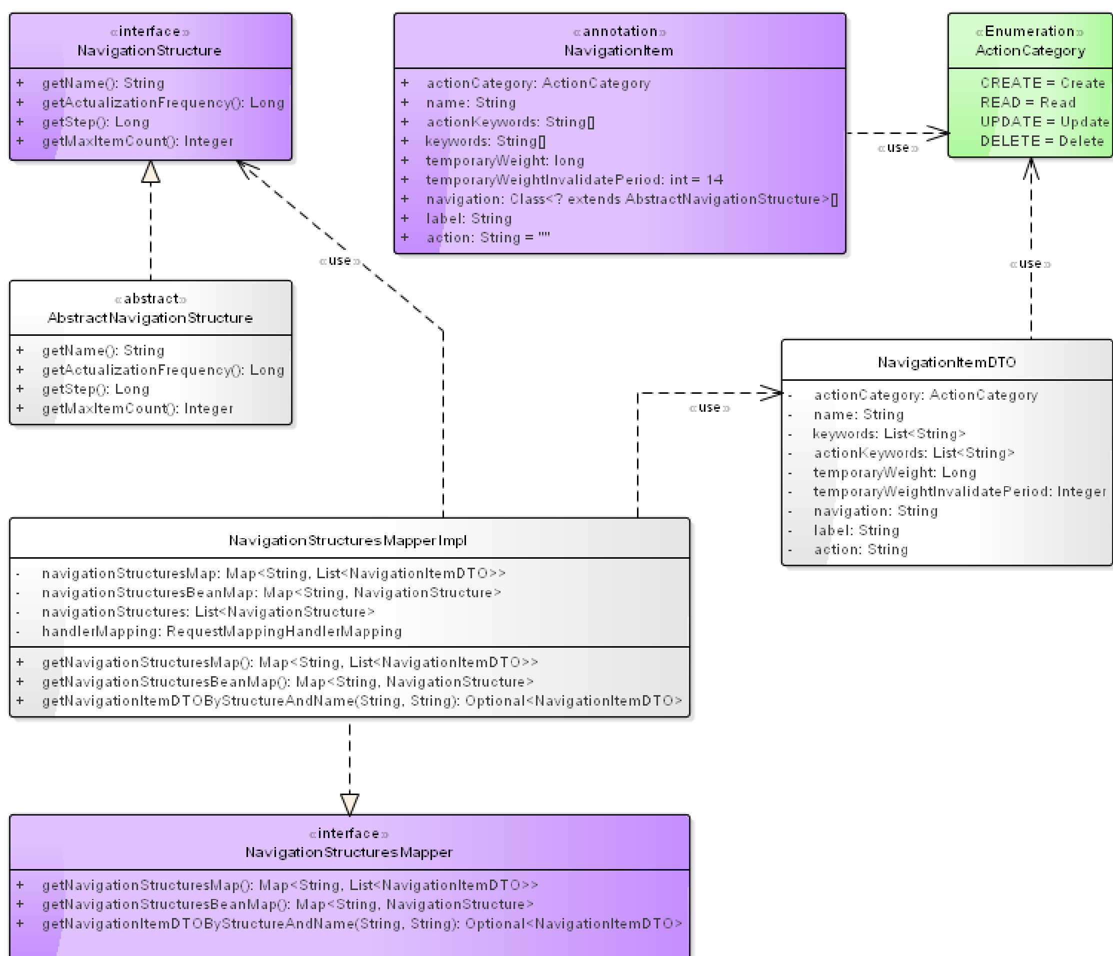
4.3 Implementace frameworku

Strukturu frameworku je možné rozdělit na několik logických části, které jsou mezi sebou spojené:

- Třídy pro definice uživatelských struktur a jejich prvků
- Třídy pro ukládání uživatelských dát
- Meta-model a třídy pro vytvoření meta-modelu
- Servisy a utility

4.3.1 Třídy pro definice uživatelských struktur a jejich prvků

Nejprve musíme nějakým způsobem zadefinovat navigační strukturu. To je zařízeno pomoci dědění od abstraktní třídy `AbstractNavigationStructure` z důvodu, že bychom poté měli nějakým způsobem kontrolovat existenci struktury pro jednotlivé prvky. Těchto struktur nebude příliš mnoho, proto vytvoření třídy nebude dělat žádný problém. Tyto třídy musí být v kontextu Springu, abychom je mohli najít. Jak jde vidět na obrázku 4.2, abstraktní třída `AbstractNavigationStructure` má naimplementovano několik metod, které vrací námi požadované hodnoty. A pokud vývojář bude chtít něco změnit, může jednoduše přepsat jakoukoliv metodu.



Obrázek 4.2. Třídy pro definice uživatelských struktur a jejich prvků

Metoda `getName()` nám vrací unikátní jméno struktury z kapitoly 3.3, což ve výchozím nastavení nám vrací jméno třídy, která bude reprezentovat navigační strukturu.

Metoda `getActualizationFrequency()` nám vrací frekvence aktualizace, která je popsána v kapitole 3.2. Má výchozí empirickou hodnotu 100. Jak už bylo řečeno, pro lepší výsledek by měl nastavovat vývojář v závislosti na účelu aplikace.

Metoda `getStep()` nám vrací krok popsaný v kapitole 3.1. Má výchozí empirickou hodnotu 100.

Metoda `getMaxItemCount()` nám vrací maximální počet prvků pod jedním rodičem. Má výchozí hodnotu 7 z kapitoly 3.5.

Navigační strukturu jde zadefinovat dalším způsobem a případně přepsat její metody:

```
@Component
public class MainMenuNavigationStructure extends
AbstractNavigationStructure {
...
}
```

Pro navigační prvky vytvářet jednotlivé třídy by bylo příliš zbytečně, navíc spojovat jejich z endpointy by taky nebylo moc jednoduché. Proto definovat navigační prvky je možné pomoci anotace `@NavigationItem`. Z obrázku 4.2 je vidět že má různé atributy.

- Atribut `actionCategory` umožňuje žádat kategorie akce která je popsána v kapitole 3.4.
- Atribut `name` odpovídá za unikátní název prvků, kapitola 3.3.
- Atribut `actionKeywords` je pole akčních klíčových slov že kterech se pak bude skládat část jména rodiče. Je popsán v kapitole 3.7. Pokud bude prázdný tak bude nastavená hodnota z kategorie akce.
- Atribut `keywords` je pole klíčových slov ze kterých se pak bude skládat další část jména rodiče. Popsán v kapitole 3.7.
- Atribut `temporaryWeight` z kapitoly 3.1 odpovídá za přechodnou váhu prvků.
- Atribut `temporaryWeightInvalidatePeriod` je počet dnů do vynulování přechodné váhy. Není to povinným atributem a má výchozí empirickou hodnotu 14 dnů. Je popsán v kapitole 3.1.
- Atribut `navigation` je pole tříd navigačních struktur do kterých patří tento navigační prvek. Ten atribut umožňuje žádat několik různých navigačních struktur.
- Atribut `label` je název navigačního prvků. Je to co vidí uživatel.
- Atribut `action` odpovídá za odkaz(URL) na který musí kliknout uživatel aby proběhla akce kliknutí na navigační prvek. Atribut není povinny a bere výchozí hodnotu z mapování Springu, ale pokud by to vývojář chtěl nějak ovlivnit, může přepsat tento odkaz.

Navigační prvek můžeme zadefinovat dalším způsobem:

```
@NavigationItem(
actionCategory = ActionCategory.CREATE,
name = "testItem",
actionKeywords = {"Create", "Manage", "Modify"},
keywords = {"test", "item", "items", "navigation"},
temporaryWeight = 50,
temporaryWeightInvalidatePeriod = 5,
navigation = MainMenuNavigationStructure.class,
label = "Test item"
)
@RequestMapping(value = "/test/item", method = RequestMethod.GET)
public Response index() {
...
```

```
}
```

Dále potřebujeme to všechno spojit a namapovat. Pro tyto účely budeme používat komponent `NavigationStructuresMapper` z obrázku 4.2.

Komponenta `NavigationStructuresMapper` bude inicializována při spuštění aplikace a bude obsahovat všichni data ohledně navigačních struktur a jejich prvků. Do komponentů se vloženy vše navigační struktury že Spring kontextu. A mapování endpoinu vezmeme z mapování Springu pomoci jeho třídy `RequestMappingHandlerMapping`. Projdeme vše endpointy který mají anotace @NavigationItem a namapujeme jejich do příslušných navigačních struktur.

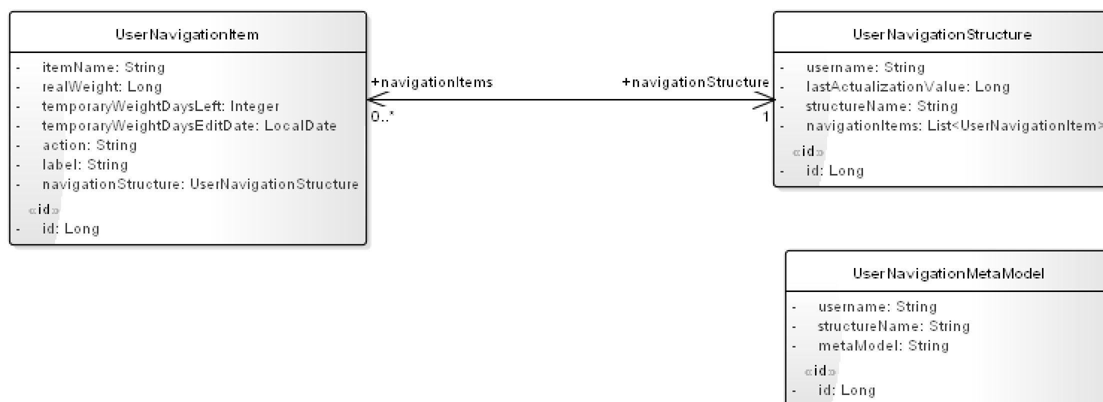
Mapování je uloženo do dvou map. Obě mapy mají jako klíč jméno navigační struktury. Jedna mapa obsahuje v sobě navigační struktury a druhá seznam příslušných navigačních prvků.

Navigační prvky je nutno nějak vytáhnout z anotace. Proto budeme používat třídu `NavigationItemDTO` na ukládání dát z anotace `@NavigationItem`, přičemž pokud jedna anotace má několik navigačních struktur, tak pro každou navigační strukturu bude vytvořen svůj `NavigationItemDTO`.

Atributy třídy `NavigationItemDTO` mají stejný význam jako atributy anotace `@NavigationItem`. Jediný rozdíl je v tom, že atribut `navigation` obsahuje unikátní jméno navigační struktury do které patří.

Takové mapování umožňuje nám snadné pohybovat po navigačním strukturám a jejich prvky. A výhodou je to že inicializace proběhne jenom jednou při spuštění aplikace.

4.3.2 Třídy pro ukládání uživatelských dát



Obrázek 4.3. Třídy pro ukládání uživatelských dát

Potřebujeme někam ukládat data jednotlivých uživatelů, přičemž nepotřebujeme ukládat všechna data, budeme ukládat jenom to co se týká konkrétních uživatelů. Data budou uložena v databázi pomocí entit z obrázku 4.3. Ostatní data můžeme zjistit pomoci komponenty `NavigationStructuresMapper`.

Entita `UserNavigationStructure` slouží pro ukládání dát pro celou navigační strukturu. Navíc je vlastníkem vztahů k `UserNavigationItem`. Z důvodů že je spojena `OneToMany` vazbou s `UserNavigationItem`, jsme schopni kontrolovat jestli byli provedeny nějaké změny v jednotlivých strukturách a případně požádat o generování nového meta-modelu. Takže můžeme získávat data pro `UserNavigationItem` které jsou společný pro celou strukturu.

Jak je vidět na obrázku 4.3, entita `UserNavigationStructure` má následující atributy:

- Atribut `username` slouží pro ukládání uživatelského jména uživatele ke kterému patří navigační struktura.
- Atribut `lastActualizationValue` je popsán v kapitole 3.2 a slouží pro ukládání sumárně hodnoty skutečných vah všech prvků který patří do navigační struktury v moment kdy byla provedena poslední aktualizace meta-modelu.
- Atribut `structureName slouží` pro úkladní názvu struktury ke které patří této data.
- Atribut `navigationItems` je seznam všech prvků této struktury pro příslušného uživatele.

Entita `UserNavigationItem` slouží pro ukládání dát o prvky pro konkrétního uživatele. Má spojení s `UserNavigationStructure` a následující atributy:

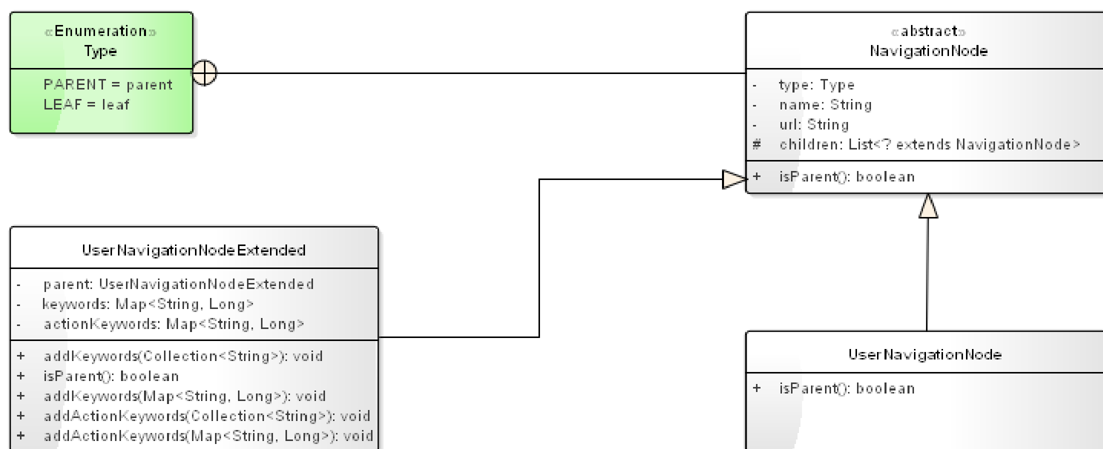
- Atribut `itemName` je unikátní název prvků, podle kterého můžeme identifikovat prvek v navigační struktuře.
- Atribut `realWeight` slouží pro ukládání skutečně váhy prvků a je popsán v kapitole 3.1. Při použití uživatelem příslušného prvků tento atribut bude inkrementován.
- Atribut `temporaryWeightDaysLeft` odpovídá za ukládání počtu dnů, do kdy by jsme měli přičítat přechodnou váhu prvků k skutečně. Popsán v kapitole 3.1.
- Atribut `temporaryWeightDaysEditDate` slouží pro ukládání data kdy byl proveden poslední odečet z počtu zbývajících dnů, aby jsme to ne odečítali několik krať denně.
- Atribut `action` slouží pro ukládání adresy prvků. De facto duplikuje atribut action třídy `NavigationItemDTO`, ale musíme to ukládat z důvodů, že pokud by tato hodnota by byla změněna v mapování, měli by jsme to detektovat a vygenerovat nový meta model.
- Atribut `label` slouží pro ukládání názvu prvků. De facto duplikuje atribut label třídy `NavigationItemDTO`, ale musíme to ukládat z důvodů, že pokud by tato hodnota by byla změněna v mapování, měli by jsme to detektovat a vygenerovat nový meta-model.

Pokud vygenerujeme meta-model, musíme ho někam ukládat aby negenerovat ho pokaždé znovu. Pro tento účel budeme používat entitu `UserNavigationMetaModel` z obrázku 4.3. Má následující atributy:

- Atribut `username` slouží pro ukládání uživatelského jména uživatele.
- Atribut `structureName` slouží pro uložení názvu navigační struktur. Spolu s atributem username umožňuje nám jednoznačně identifikovat meta model pro uživatele a navigační strukturu.
- Atribut `metaModel` slouží pro ukládání samotného meta modelu serializovaného ve json formátu. Při nutnosti generování nového meta modelu hodnota atributů bude změněna. Jinak pokud potřebujeme meta model pro uživatele a navigační strukturu nebudeme muset to generovat znovu, stačí vzít hodnotu tohoto atributů a deserializovat její.

4.3.3 Meta model a třídy pro vytvoření meta modelu

Meta model by neměl obsahovat zbytečně atributy, které nepotřebujeme pro zobrazení uživatelů, ale když budeme meta model generovat potřebovali bychom vědět několik atributů navíc.



Obrázek 4.4. Třídy pro vytvoření meta-modelu

Meta-model je stromová struktura a sestavena z objektů typu `UserNavigationNode`. Objekty typu `UserNavigationNodeExtended` budou složit pro postavení struktury meta modelu, a lze je poté snadno transformovat do `UserNavigationNode`. Třídy jsou zobrazeny na obrázku 4.4.

Třída `UserNavigationNode` má další atributy:

- Atribut `type` umožňuje detekovat jestli je to rodič nebo potomek. Může nabývat hodnoty rodič nebo potomek. Tak to uděláno aby jsme mohli sestavovat celou strukturu meta-modelu z objektů jenom jedné třídy, což nám zjednoduší práce a zdroje při serializace a deserializace struktury.
- Atribut `name` odpovídá za název uzlu. Je to co uvidí uživatel.
- Atribut `url` je odkaz na příslušný endpoint.
- Atribut `children` obsahuje v sobě všichni potomky tohoto uzlu.

Třída `UserNavigationNodeExtended` má navíc ještě 3 atributů:

- Atribut `parent` pro jednodušší pohybování ve stromů.
- Atributy `keywords` a `actionKeywords` pro pojmenování rodičovských uzlu.

Pro vytvoření správné struktury budeme používat třídu `NodeTreeBuilder`. Tato třída má dva konstantní atributy *maximální počet prvků ve vyšším úrovni* [Obrázek 3.4] a *počet prvků ve všech ostatních úrovních*, které jsme nastavujeme pomoci argumentů konstruktoru třídy.

Navíc má atribut `nodes` typu `List<UserNavigationNodeExtended>` a je prázdný. Tento atribut budeme používat pro ukládání struktury.

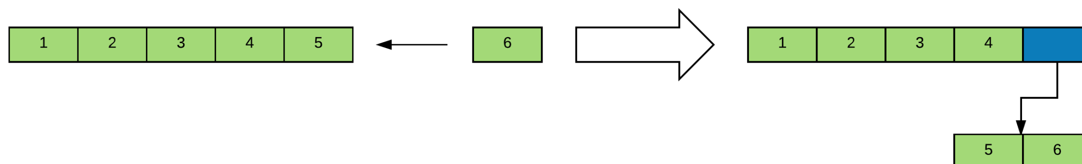
Třída `NodeTreeBuilder` má dvě metody:

- Metoda `add(UserNavigationNodeExtended node)` slouží pro přidání předem uspořádaných prvků do struktury.
- Metoda `build()` bude generovat meta-model a vrací stromovou strukturu typu `List<UserNavigationNode>`, kde tento `List` obsahuje takový počet prvků, který jsme žádali v konstruktoru třídy jako maximální počet prvků ve vyšším úrovni.

Z důvodů že budeme přidávat prvky v pořadí zmenšení jejich aktuální váhy, vždy budeme mít na začátku seznamu prvky z největší vahou a cesta do těchto prvků bude nejkratší.

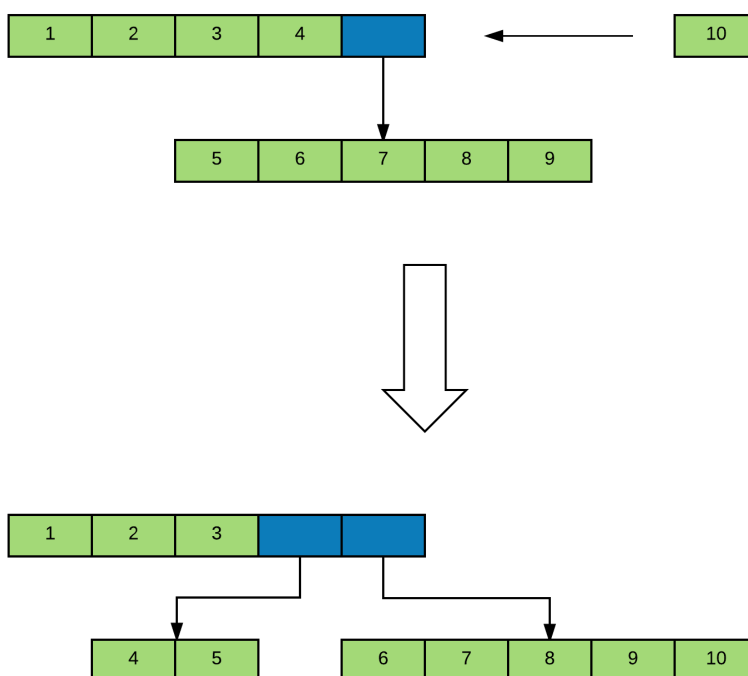
Přidávání prvků do struktury meta modelu bude proběhat přidáním tohoto prvků do seznamu, pokud nedosáhneme limit. Po dosažení limitu, budeme musit vytvořit

rodičovský uzel. Jako potomky jemu nastavíme poslední prvek seznamu a ten prvek který chceme vložit. A dáme ten rodičovský uzel na místo posledního prvků seznamu, viz obrázek 4.5.



Obrázek 4.5. Přidání prvků do seznamu který dosáhl svůj limit

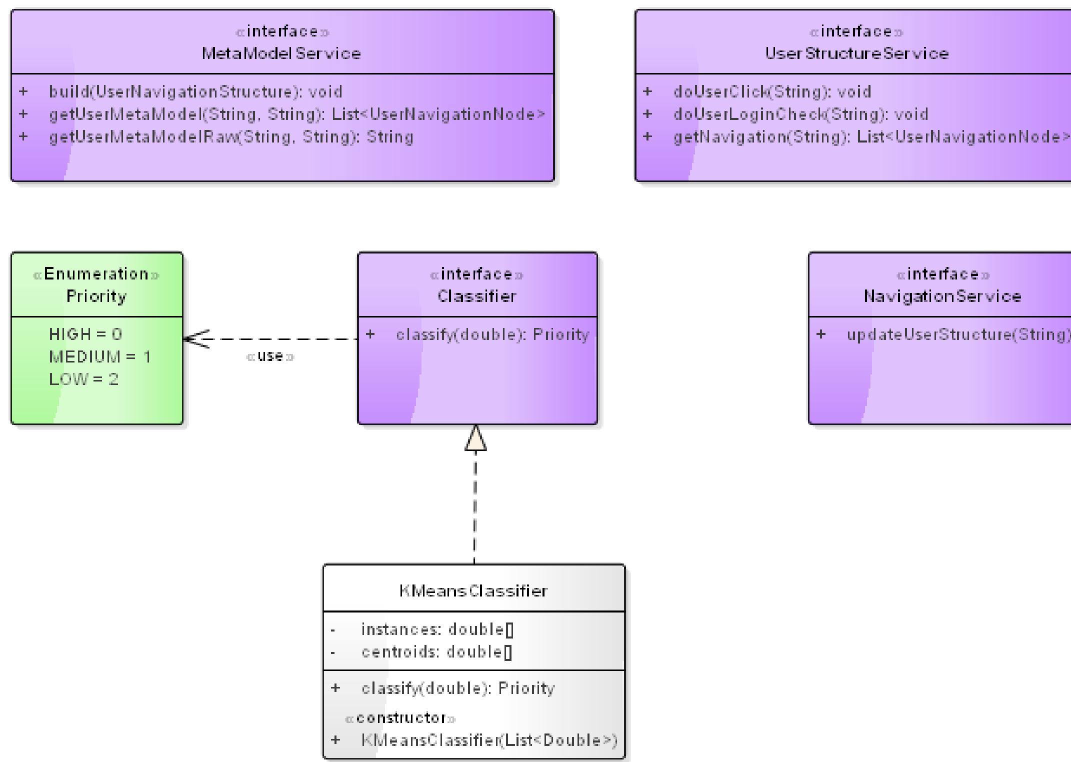
Někde se může nastat situace kdy podseznam taky dosáhne svého limitu. A budeme musit vytvořit dalšího rodiče. Ale z důvodů že máme na začátku seznamu prvky s nejvyšší vahou a cesta do nich má být nejkratší, ale vkládáme prvky které mají váhu méně než jakýkoliv prvek že seznamu, při vytvoření dalšího rodiče musíme vzít prvek z nejvyšší vahou z předchozího rodiče a dát ho do nového rodiče a ten prvek který chceme vkládat už dáme na konec seznamu předchozího rodiče, viz obrázek 4.6.



Obrázek 4.6. Přidání prvků do podseznamu který dosáhl svůj limit

Pokud už všichni prvky v jedněm seznamu rodiči, tak déme na úroveň hlubší a děláme to samé. Když přidáme do builderu všichni prvky které jsme chtěli a zavoláme metodu `build()`, tak na začátku builder projde všichni uzly toho stromů od nejnižších a agreguje klíčové slova do rodičů. A pak půjde směrem z hora dolů a transformuje uzly třídy `UserNavigationNodeExtended` do třídy `UserNavigationNode`, přičemž pokud to bude rodičovský uzel, tak vygeneruje název toho uzlu, a setřídí všichni potomky jak to popsáno ve kapitole 3.5, to znamená, že prvky s nejvyšší prioritou budou na začátku a na konce seznamu.

4.3.4 Servisy a utility

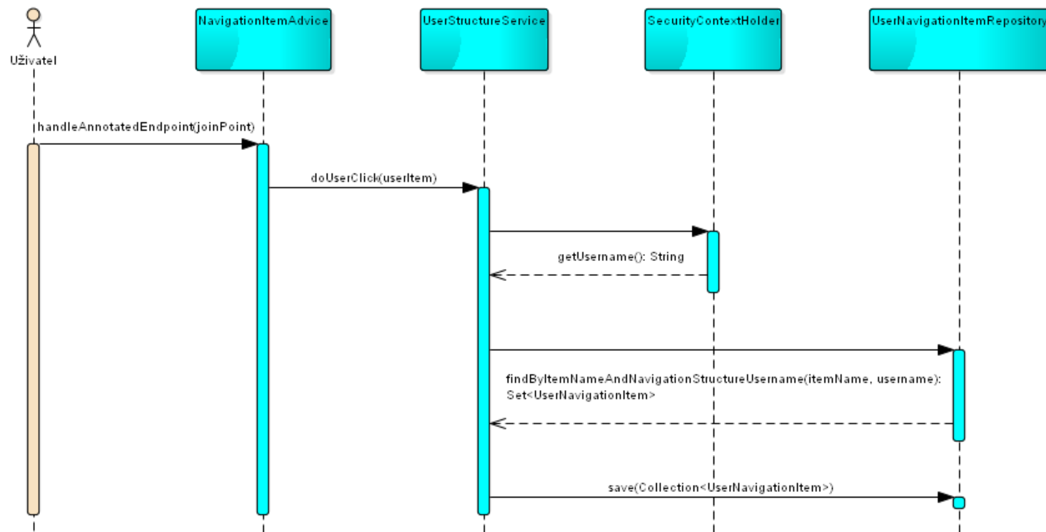


Obrázek 4.7. Servisy a utility

Máme několik servis které komunikují mezi sebou a každá z nich odpovídá za určitou část [Obrázek 4.7]

Servisa `UserStructureService` je jediná servisa se kterou by měl komunikovat vývojář a která používá aspekty. Servisa ví o ostatních servisu a používá je pro své účely.

Metoda `doUserClick` servisy `UserStructureService` má parametrem unikátní název prvků po kterému byla provedena akce. Vezme ze bezpečnostního Spring kontextu uživatelské jméno běžného uživatele a najde všichni prvky s názvem, který dostane jako argument a který patří uživatelů a zvýší počet jejich použití o jeden, viz obrázek 4.8.



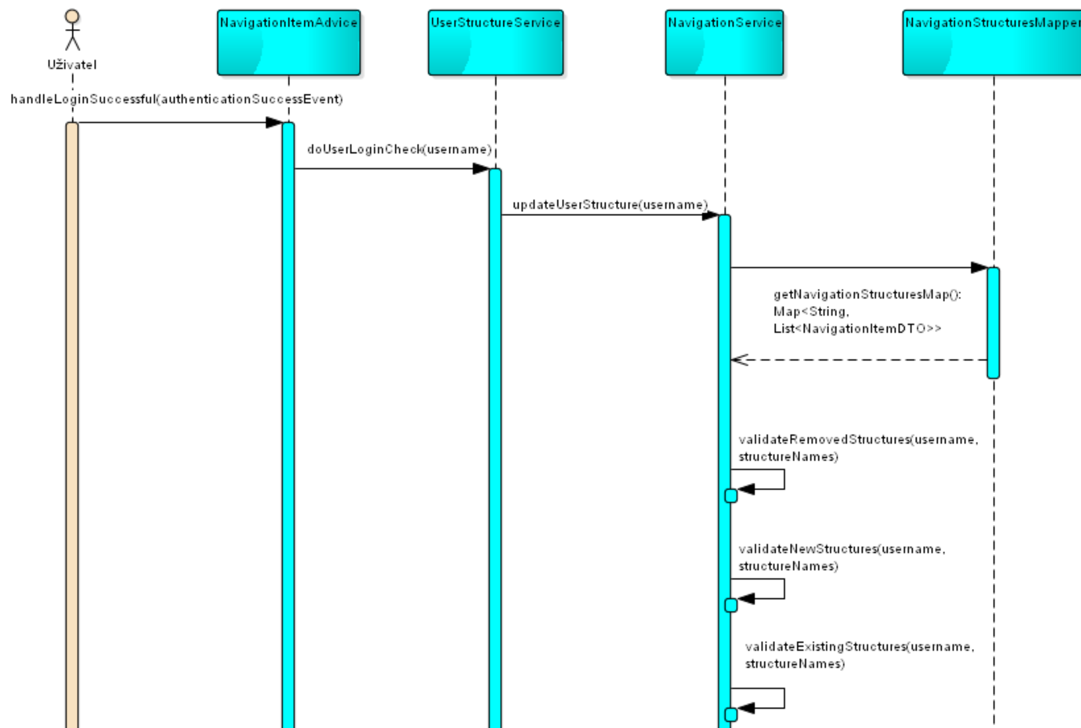
Obrázek 4.8. Sekvenční diagram vyvolání uživatelem jednotlivého endpointu

Metoda `doUserClick` je volaná aspektem podle šablony

```
@annotation(com.wikbit.annotation.NavigationItem) && execution(* *(..))
```

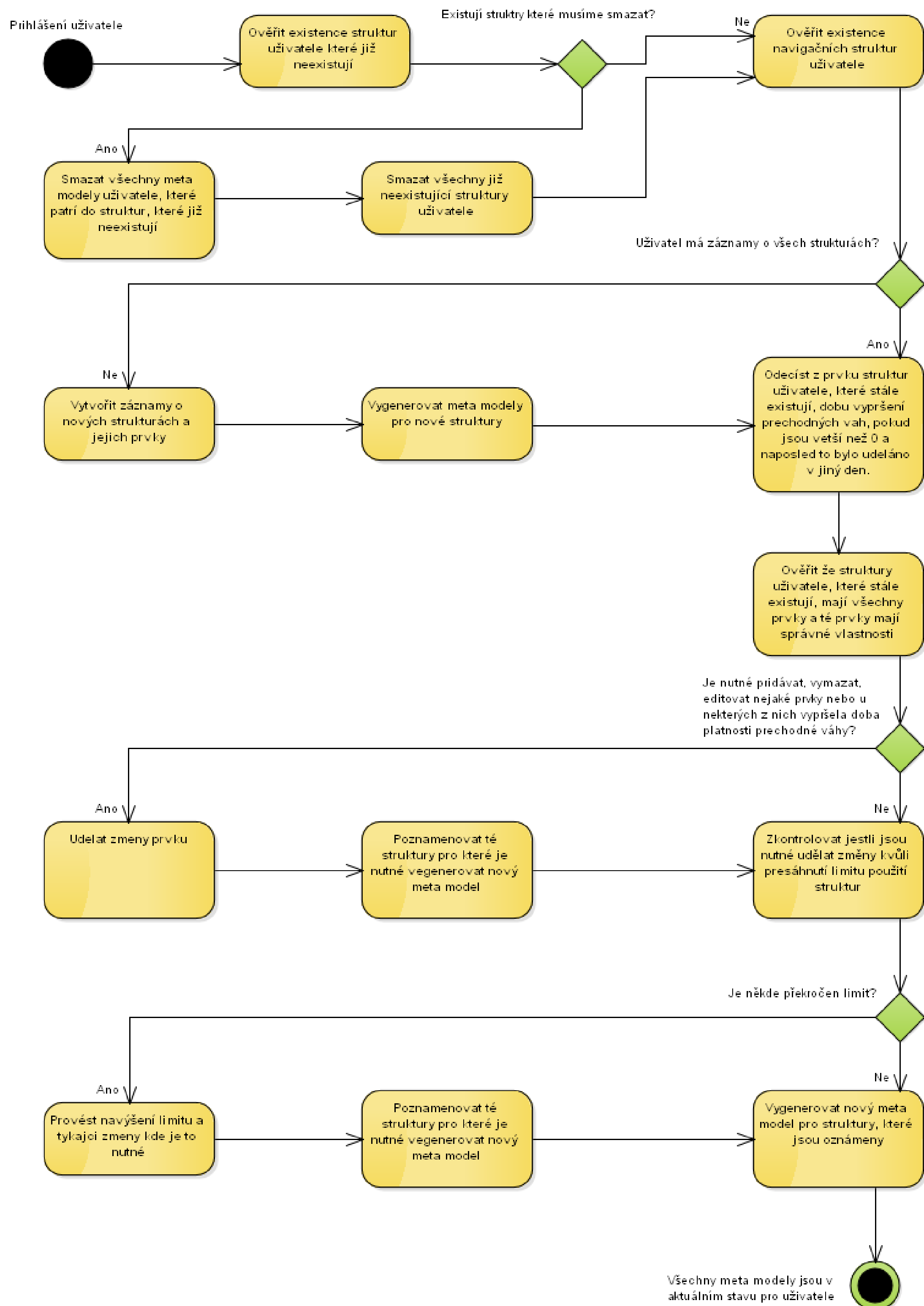
Metoda `getNavigation` servisy `UserStructureService` vrací meta-model podle názvu navigační struktury a uživatelského jména, které zjistí že bezpečnostního Spring kontextu. Sebere všichni potřebný data a zavolá `MetaModelService`.

Metoda `doUserLoginCheck` servisy `UserStructureService` má parametrem uživatelské jméno, a volá NavigationService při přihlášení uživatele, aby `NavigationService` servisa zkontrolovala, jestli máme nějaké změny ve struktuře aplikace nebo jestli nenastal čas obnovit meta model [Obrázek 4.9].



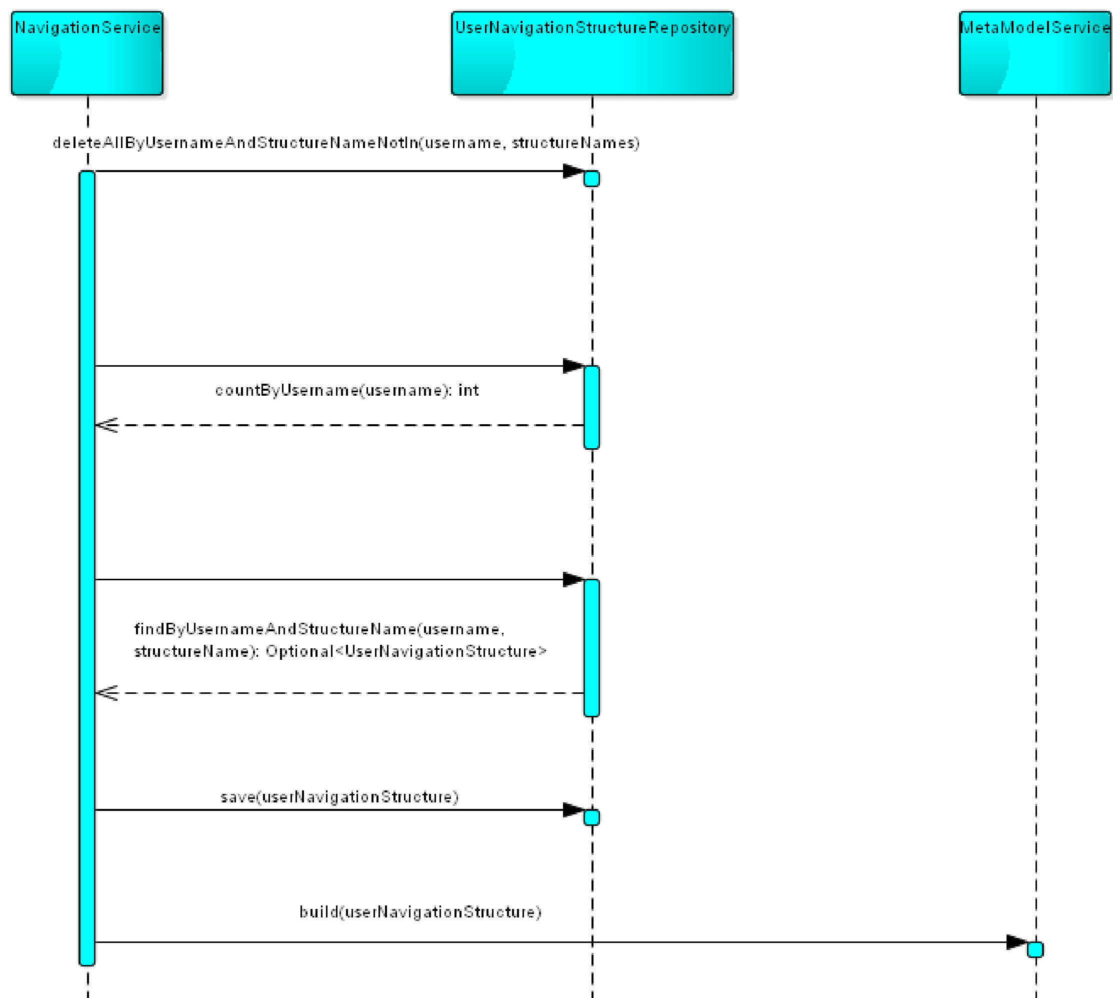
Obrázek 4.9. Ověření nutnosti generování nových meta-modelu

`NavigationService` servisa se zabývá tím, že kontroluje, jestli byly nějaké změny ve strukturách, nebo jestli nenastal čas kdy jsme překročili další hranice, kdy musíme obnovit meta-model. Dále kontroluje jestli nevypršela doba platnosti přechodné váhy pro nějaký prvek, a v případě nutnosti aktualizuje uživatelská data a vygeneruje meta-model. Algoritmus ověření je popsán na obrázku 4.10.



Obrázek 4.10. Kontrola změn a nutnosti generování meta-modelu

Pokud během této kontroly zjistíme že nutné uložit nějaké změny, nebo vygenerovat nový meta-model, nebo uložit změny a vygenerovat meta-model pro nějakou navigační strukturu tak ta servisa to udělá a zavolá příslušnou akce [Obrázek 4.11].



Obrázek 4.11. Ukládání dát a generování nového meta-modelu v NavigationService servise

Servisa `MetaModelService` odpovídá za generování meta-modelu a načtení meta-modelu.

Metoda `getUserMetaModelRaw` servisy `MetaModelService` načte ze databáze meta-model podle uživatelského jména a názvu navigační struktury a vrátí meta-model ve serializovanem tvaru.

Metoda `getUserMetaModel` servisy `MetaModelService` dělá to samé co metoda `getUserMetaModelRaw`, ale navíc deserializuje meta-model a vrátí nám už deserializovanou verze.

Nejzajímavější je metoda build, ze `MetaModelService` servisy. Ta dostává jako vstupní parametr `UserNavigationStructure` (navigační strukturu uživatele) s aktuální data obnovené v předchozím kroku `NavigationService` servisou a z ní vygeneruje meta-model a uloží ho do databáze.

Generování meta-modelu probíhá tímto způsobem:

- Na začátku vezmeme všichni prvky navigační struktury uživatele a setřídíme jejich podle aktuálních vah sestupně.

- Vytvoříme klasifikátor, instance třídy `KMeansClassifier` z obrázku 4.7. Jako parametr do konstruktoru dáme seznam všech navigačních prvků uživatele z této navigační struktury. `KMeansClassifier` vezme této data, zvolí tři centroidy, každý z nich bude odpovídat za jednou prioritu (Vysoká, Střední, Nízká) a natrénuje jejich že vstupních dát pomoci algoritmu K-means. A teď už můžeme klasifikovat jednotlivé prvky podle aktuální váhy. To jsme udělali aby bylo možné vidět kde je ta hranice mezi prvky.
- Projdeme všichni prvky a vložíme jejich do mapy kde klíčem je priorita a hodnotou seznam prvků, který patří do příslušné priority.
- Dál musíme řešit kolik míst ve vyšším úrovně navigační struktury zarezervujeme pro prvky z vysokou prioritou a kolik pro ostatní. Proto pokud maximální limit na šířku je 2 (z menším limitem žádnou navigace neuděláme) tak jeden bude rezervovat pro prvky s vysokou prioritou a druhý pro ostatní. Pokud maximální limit je 3 a více tak pro prvky s vysokou prioritou bude rezervováno limit - 2 pozic, nebo počet pozic bude stejným jako počet prvků s vysokou prioritou, pokud je menší než limit - 2. Ostatní pozice budou pro prvky se střední a nízkou prioritou.
- Pomoci dvou instanci třídy `NodeTreeBuilder`, která je popsán, vytvoříme struktury pro prvky z vysokou prioritou a pro ostatní a spojíme do jednoho seznamu.
- Na konce setřídí ten výsledný seznam způsobem, který je popsán v kapitole 3.5, to znamená že prvky s nejvyšší prioritou budou na začátku a na konce seznamu, a uloží ho do databáze v `UserNavigationMetaModel`.

4.4 Požadavky pro použití frameworku

- Aplikace používá Java verze 8
- Aplikace musí být založena na Spring frameworku
- Aplikace musí používat pro bezpečnost Spring security
- Musí být nakonfigurováno připojení k databáze
- Databáze musí mít následující strukturu z obrázku 4.12 pro ukládání entit `UserNavigationItem`, `UserNavigationMetaModel`, `UserNavigationStructure`.

Obrázek 4.12. Struktura databáze

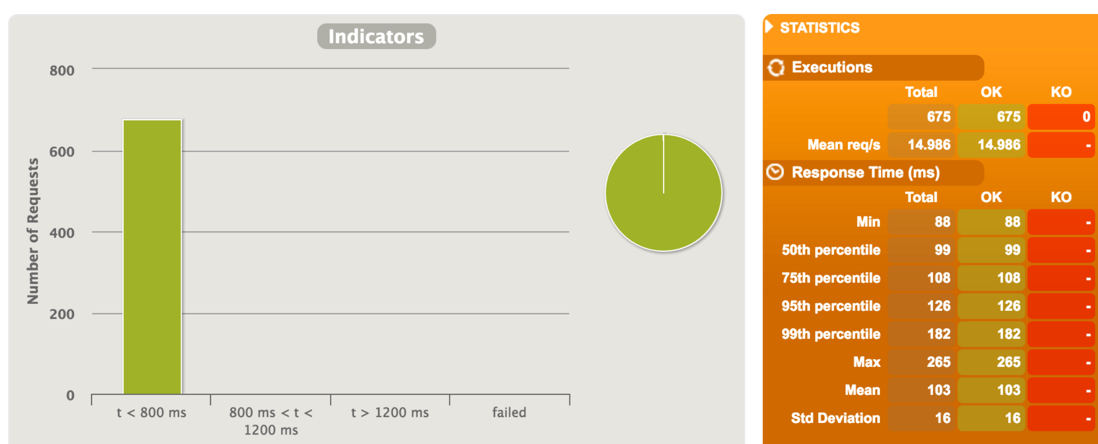
A potřebujeme mít tři sekvence v databáze, které se budou jmenovat:

- `user_navigation_structure_seq`
- `user_navigation_item_seq`
- `user_navigation_meta_model_seq`

Kapitola 5
Testování

Testování bylo provedeno pomoci frameworku Gatling. Ten se slouží pro zátěžové testování. On se posílá požadavky na náhodně endpointy. Takový přístup nám umožňuje otestovat dvě cíle.

První je to že systém stabilně funguje při velkých zatížení. Nastavil jsem ten test tak aby posílal 15 požadavků za vteřinu během 45 vteřin. Což nám dokázalo že systém funguje stabilně a nedošlo k žádným poruchám.



Executions	Total	OK	KO
	675	675	0
Mean req/s	14.986	14.986	-

Response Time (ms)	Total	OK	KO
Min	88	88	-
50th percentile	99	99	-
75th percentile	108	108	-
95th percentile	126	126	-
99th percentile	182	182	-
Max	265	265	-
Mean	103	103	-
Std Deviation	16	16	-

Obrázek 5.1. Výsledek testování v Gatling

Jak je vidět na obrázku 5.1, všichni požadavky proběhlé úspěšně a odpověď aplikace byla rychlá.

Druhá a hlavní cíl je naplnit aplikace uživatelskými data a podívat se jak bude chovat navigační struktura.

Pro testování jsem zvolil maximální počet prvků 5 pro jednoho rodiče. Dohromady mám 24 prvků v jedné struktuře. Pro výpočet cesty použil jsem další princip, té prvky co jsou v nejvyšším úrovni má délku cesty 0, pro ostatní cesta bude záviset na počtu rodičů které musíme projít aby dosáhnout cíle.

Název prvků	Aktuální váha	Cesta
deleteStatistic	49.0625	0
editCustomer	47.0625	1
addCategory	46.0625	1
deleteCategory	46.0625	1
addAdvertisement	46.0625	1
viewStatistic	46.0625	1
deleteOrder	45.0625	1
addOrder	45.0625	1
viewAdvertisement	44.0625	1
editOrder	44.0625	1
addProduct	43.0625	1
editAdvertisement	43.0625	1
viewCategory	42.0625	1
viewOrder	42.0625	1
deleteAdvertisement	41.0625	1
deleteProduct	40.0625	1
editProduct	40.0625	2
addStatistic	39.0625	2
editCategory	39.0625	2
editStatistic	38.0625	2
viewCustomer	37.0625	2
deleteCustomer	37.0625	2
viewProduct	37.0625	2
addCustomer	35.0625	2

Tabulka 5.1. Cesta do prvků se základní data

Jak je vidět v tabulce 5.1, data jsou praktické stejně, a čím větší prvek má váhu, tým kratší cesta do toho prvků.

Zvolíme 3 prvky z nejmenší vahou, spustíme test jen pro té 3 prvky a podíváme se jak se změní cesta do těchto prvků.

Název prvků	Aktuální váha	Cesta
viewProduct	294.125	0
addCustomer	283.125	0
deleteCustomer	249.125	0
deleteStatistic	63.125	1
editCustomer	61.125	1
addCategory	60.125	1
deleteCategory	60.125	1
addAdvertisement	60.125	1
viewStatistic	60.125	1
deleteOrder	59.125	1
addOrder	59.125	2
viewAdvertisement	58.125	2
editOrder	58.125	2
addProduct	57.125	2
editAdvertisement	57.125	2
viewCategory	56.125	2
viewOrder	56.125	2
deleteAdvertisement	55.125	2
deleteProduct	54.125	2
editProduct	54.125	2
addStatistic	53.125	2
editCategory	53.125	2
editStatistic	52.125	2
viewCustomer	51.125	2

Tabulka 5.2. Cesta do prvků s zvýrazněním třech prvků

Jak je vidět z tabulky 5.2 už je použití jednotlivých prvků je výraznější. Té prvky který jsme zvolili teď jsou nahoře a mají nejkratší cestu. Takovou cestu mají vše tři ještě z důvodů že rozdíl mezi ní a ostatní prvky je docela velký. Ještě jedna věc kterou jde vidět je to že aktuální váha se zvýšila pro prvky který jsme nepoužívali, a to je z důvodů že zvýšila celková reální váha celé struktury a to znamená že zvýšil se koeficient. Přechodnou váhu máme platnou proto spolu s koeficientem to ovlivňuje aktuální váhu těch prvků pro které ona platí, což v našem případě všichni prvky.

Kapitola 6
Instalace

- Vložte CD do mechaniky.
- Rozbalte obsah archívu AdaptiveApplication.zip na svůj disk.
- Pomoci IntelliJ IDEA IDE, nebo nějakou jiné která se Vám líbí nejvíc, naimportujte projekt z již existující zdroje z modele Gradle.
- Nakonfigurujte databáze a aplikace podle požadavků z kapitoly 4.4.
- Spusťte Gradle zadání build.
- Spustit Spring boot aplikace z module-core která se jmenují BachelorThesisApplication.
- Hotovo.

Kapitola 7
Závěr

Bakalářská práce byla vypracována podle přiděleného zadání katedrou počítačů fakulty elektrotechnické Českého vysokého učení technického v Praze v rámci studijního programu Softwarové technologie a management obor Softwarové inženýrství.

Zadáním bakalářské práce bylo prostudovat již existující řešení pro vytvoření adaptivní struktury aplikace Java EE s použitím aspektově-orientovaného přístupu, zanalyzovat problémy na které můžeme narazit, naimplementovat řešení a otestovat ho.

Byl vytvořen framework který nám umožní generovat strukturu pro jednotlivých uživatelů v závislosti na jejich chování. Je flexibilní a konfigurovatelný. Umožňuje programátorům používat ho pro různé účely.

Testování proběhlo úspěšně. Během testování bylo zjištěno, že cesta do nejpoužívanějších prvků ve vygenerované frameworkem struktuře je nejkratší a aplikace je odolná proti velkému počtu uživatelů.

Literatura

[1] "Abowd. *"Handheld and Ubiquitous Computing: First International Symposium.* In: "Berlin: "Springer Berlin Heidelberg", ISBN "978-3-540-48157-7".
`"http://dx.doi.org/10.1007/3-540-48157-5_29"` .

[2] "Hanumansetty. "Model based approach for context aware and adaptive user interface generation".

[3] "Chen. *"A survey of context-aware mobile computing research"*. . "Technical Report TR2000-381.

[4] "Schmidt. *"Handheld and Ubiquitous Computing: First International Symposium.* In: "Berlin: "Springer Berlin Heidelberg", ISBN "978-3-540-48157-7".
`"http://dx.doi.org/10.1007/3-540-48157-5_10"` .

[5] "Kiczales. *"ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä.* In: "Berlin: "Springer Berlin Heidelberg", ISBN "978-3-540-69127-3".
`"http://dx.doi.org/10.1007/BFb0053381"` .

[6] "Wikipedia". *"Aspect-oriented programming — Wikipedia, The Free Encyclopedia"*. "2016".
`"https://en.wikipedia.org/w/index.php?title=Aspect-oriented_programming&oldid=718586186"` .

[7] "Turek Tomáš". "Využití aspektově-orientovaného přístupu pro tvorbu adaptivních uživatelských rozhraní". "2015",

[8] *"Spring"*.
`"http://docs.spring.io/"` .

[9] *"Spring AOP documentation"*.
`"http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html"` .

[10] "Andy Crestodina". "Are You Making These Common Website Navigation Mistakes?".

[11] "McLeod. "Serial Position Effect".

[12] "Wikipedie". *"Scala (programovací jazyk) — Wikipedie: Otevřená encyklopedie"*. "2016".
`"cs.wikipedia.org/w/index.php?title=Scala_(programovac%C3%AD_jazyk)&oldid=13624468"` .

[13] *"The Scala Programming Language"*.
`"http://www.scala-lang.org/"` .

[14] "Fielding. *"Architectural styles and the design of network-based software architectures"*. Disertační práce, "University of California.

[15] "Wikipedia". *"Representational state transfer — Wikipedia, The Free Encyclopedia"*. "2016".

"en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=721854851" .

[16] "Wikipedia". *"Object-relational mapping — Wikipedia, The Free Encyclopedia"*. "2016".
"https://en.wikipedia.org/w/index.php?title=Object-relational_mapping&oldid=719087990" .

[17] *"Hibernate. Everything data."*.
"http://hibernate.org/" .

[18] *"Spring Data JPA"*.
"http://projects.spring.io/spring-data-jpa/" .

[19] *"Spring Security"*.
"http://projects.spring.io/spring-security/" .

Příloha A
Zadání práce

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Nikita Mishchenko**

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: **Aspektově orientovaný vývoj adaptivní struktury aplikace pro Java EE aplikace**

Pokyny pro vypracování:

Prostudujte existující nástroje pro podporu aspektově orientovaného programování (AOP).
Seznamte se s vývojem aplikací pro Java EE. Nastudujte a zpracujte rešerši o adaptivní struktuře aplikace s ohledem na její kontext.
Dále proveďte rozbor problémů při realizaci uživatelského rozhraní (UI).
Navrhněte a implementujte framework, který využije kontextové informace při generování adaptivního uživatelského rozhraní (AUI) s ohledem na AOP.
Výsledná implementace musí být snadno rozšiřitelná. Implementaci otestujte na demonstrační aplikaci. Zhodnoťte výhody a možná omezení řešení.

Seznam odborné literatury:

1. ŠEBEK, J. and K. RICHTA. Aspect-oriented User Interface Design for Android Applications [online]. In: DATESO 2015. Databases, Texts, Specifications, and Objects 2015, Nepřívěc u Sobotky, Jičín, 2015-04-14/2015-04-16. Praha: MATFYZPRESS, vydavatelství Matematicko-fyzikální fakulty UK, 2015, pp. 121-130. CEUR Workshop Proceedings. vol. 1343. ISSN 1613-0073. ISBN 9788073782856. Available from: http://www.cs.vsb.cz/dateso/2015/
2. Cerny, Tomas, et al. "Aspect-driven, data-reflective and context-aware user interfaces design." ACM SIGAPP Applied Computing Review 13.4 (2013): 53-66. (http://dl.acm.org/citation.cfm?id=2577561)
3. HANUMANSETTY, Reena Gowri. Model based approach for context aware and adaptive user interface generation. 2004. PhD Thesis. Virginia Tech. (https://vtechworks.lib.vt.edu/handle/10919/10087)

Vedoucí: Ing. Jiří Šebek

Platnost zadání: do konce letního semestru 2016/2017

prof. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 14. 1. 2016

Příloha B
Zkratky

AOP Aspektově orientované programování
JPM Join point model
AAS Adaptive Application Structure
AUI Adaptivní uživatelské rozhraní
IoC Inverse of control
UI User Interface
REST Representational State Transfer
URL Uniform Resource Locator

Příloha C Obsah CD

- bakalarskaPraceNikitaMishchenko.pdf - elektronická verze bakalářské práce
- bakalarska_prace - použité materiály, obrázky, atd
- AdaptiveApplication.zip - implementace