



## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

<b>Název:</b>	Serverový backend Android aplikace pro podporu rodin v rozvodovém řízení
<b>Student:</b>	Iaroslav Kolodka
<b>Vedoucí:</b>	Ing. Jiří Hunka
<b>Studijní program:</b>	Informatika
<b>Studijní obor:</b>	Webové a softwarové inženýrství
<b>Katedra:</b>	Katedra softwarového inženýrství
<b>Platnost zadání:</b>	Do konce letního semestru 2020/21

### Pokyny pro vypracování

Cílem této práce je realizovat serverový backend android aplikace pro podporu rodin v rozvodovém řízení. Frontend (mobilní aplikaci) řeší souběžně ve spolupráci s autorem této práce bakalářský student Martin Beran.

Postupujte v těchto krocích:

1. V návaznosti na Softwarový Týmový Projekt analyzujte současný stav návrhu backendu spolu se současným stavem fragmentu implementace.
2. Na základě analýzy navrhnete vhodné úpravy tak, aby byl dosažen kvalitnější výsledek a zároveň byly splněny požadavky frontendové části (mobilní aplikace).
3. Pro korektní spolupráci s Martinem Beranem vhodně dokumentujte API.
3. Implementujte serverový backend dle analýzy a návrhu.
4. Při implementaci věnujte řádnou pozornost testům - navrhnete a aplikujete vhodné testy.
5. Zhodnoťte použitelnost výsledné implementace a navrhnete vhodné budoucí kroky (např. rozšíření, technologické směřování, apod.)

### Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.  
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.  
děkan

V Praze dne 14. února 2020





**FAKULTA  
INFORMAČNÍCH  
TECHNOLGIÍ  
ČVUT V PRAZE**

Bakalářská práce

## **Serverový backend Android aplikace pro podporu rodin v rozvodovém řízení**

*Iaroslav Kolodka*

Vedoucí práce: Ing. Jiří Hunka

4. června 2020



---

## Poděkování

Rád bych poděkoval všem lidem, bez kterých by tato práce nemohla vzniknout. Hlavně bych chtěl poděkovat vedoucímu práce Ing. Jiřímu Hunkovi, který měl vždycky čas na moudrou radu. Dále děkuji manažerovi tohoto projektu Oldřichu Malcovi, který měl vždy čas na pomoc, když jsem narazil na jakýkoliv problém. Zvláštní poděkování patří mému kolegovi Martinu Beranovi, který současně pracoval na frontdendu aplikace a vždycky srozumitelně popisoval požadavky na API serveru.



---

# Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. Dále prohlašuji, že jsem s Českým vysokým učení technickým v Praze uzavřel dohodu, na jejímž základě se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ustanovení § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisu.

V Praze dne 4. června 2020

.....

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2020 Iaroslav Kolodka. Všechna práva vyhrazena.

*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

### **Odkaz na tuto práci**

Kolodka, Iaroslav. *Serverový backend Android aplikace pro podporu rodin v rozvodovém řízení*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020. Dostupný také z WWW: <https://github.com/Iaroslav-K/thesis-bp>.



---

# Abstrakt

Tato bakalářská práce se zabývá realizací serverového backendu Android aplikace pro podporu rodin v rozvodovém řízení. Implementace se provádí na základě již existujícího návrhu a částečné implementace, které byly provedeny v rámci předmětů BI-SP1 a BI-SP2 bakalářského studia vyučovaných na FIT ČVUT v Praze. Zmíněné předměty jsou zaměřené na studium pomocí praktického vyzkoušení analýzy, návrhu a realizace rozsáhlejšího softwarového systému. Pro dosažení lepších výsledků byly navrženy vhodné úpravy podle požadavků frontedové části aplikace. Backend aplikace je napsán v jazyce Kotlin. Pro návrh REST API byl zvolen framework Spring. Pro testování aplikace byly zvoleny frameworky JUnit 5 a funkce, které poskytuje framework Spring. Za účely návrhu bezpečnosti server využívá protokoly HTTPS a OAuth 2.0. Pro dokumentaci REST API byl zvolen framework Swagger. Byly navrženy vhodné budoucí kroky, které budou implementovány po dokončení této práce.

**Klíčová slova** serverový backend, Kotlin, REST, Spring, rozvodové řízení

---

# Abstract

This bachelor thesis deals with the implementation of a server backend Android application to support families in divorce proceedings. Implementation is carried out based on an existing design and partial implementation, which were carried out within the subjects BI-SP1 and BI-SP2 of bachelor's studies taught at FIT CTU in Prague. To achieve better results, suitable modifications were proposed according to the requirements of the frontend part of the application. The backend application is written in the Kotlin language. The Spring framework was chosen to design the REST API. The JUnit frameworks and the functionality provided by the Spring framework were chosen to test the application. For security design purposes, the server uses the HTTPS and OAuth 2.0 protocols. The Swagger framework was chosen for the REST API documentation. Appropriate future steps have been proposed and will be implemented after the completion of this thesis.

**Keywords** server-side backend, Kotlin, REST, Spring, divorce proceedings

---

# Obsah

Úvod	1
<b>1 Teoretická část</b>	<b>3</b>
1.1 Pravidla frameworku Angular pro Git	3
1.1.1 Typ	4
1.1.2 Rozsah	5
1.1.3 Předmět	5
1.1.4 Tělo	5
1.1.5 Zápatí	5
1.2 Použitý jazyk programování	5
1.3 Nástroj pro automatizaci sestavování programu	6
1.4 Nástroj pro vývoj podnikových aplikací	6
1.4.1 Spring framework	6
1.4.2 Spring Boot	8
1.4.3 Spring Security	8
1.4.4 Spring Data	8
1.4.5 Spring Web Services	8
1.5 Testování	9
1.5.1 Spring	9
1.5.2 JUnit	9
1.5.3 JaCoCo	9
1.5.4 IntelliJ IDEA	10
1.6 Dokumentace	10
1.7 Použité databáze	10
1.7.1 H2	10
1.7.2 PostgreSQL	10
1.8 Nástroj pro analýzu rozsahu implementace	11
<b>2 Analýza</b>	<b>13</b>

2.1	Předmět Softwarový týmový projekt 1 . . . . .	13
2.2	Předmět Softwarový týmový projekt 2 . . . . .	14
2.3	Doménový model . . . . .	14
2.4	Analýza předešlého návrhu . . . . .	14
2.4.1	Registrace a přihlášení do systému . . . . .	15
2.4.2	Kalendář . . . . .	15
2.4.3	Alimenty . . . . .	17
2.4.4	Knihy potřeb dítěte . . . . .	18
2.4.5	Účtenky . . . . .	19
2.4.6	Požadavky na změny . . . . .	19
2.4.7	Historie změn . . . . .	20
2.5	Analýza předešlé implementace . . . . .	21
2.5.1	Implementované entity . . . . .	22
2.5.2	Dokumentace API . . . . .	23
2.5.3	Profily . . . . .	24
2.5.4	Databáze . . . . .	24
2.6	Analýza požadavků na změny frontendové části aplikace . . . . .	24
2.6.1	Interval . . . . .	25
2.6.2	Alimenty . . . . .	25
2.6.3	Pečovatelské dny . . . . .	26
2.6.4	Oznámení . . . . .	27
2.7	Analýza bezpečnosti . . . . .	28
2.7.1	Role . . . . .	28
2.7.2	Autorizace . . . . .	28
2.8	Analýza testování . . . . .	29
2.9	Průběžná integrace . . . . .	29
<b>3</b>	<b>Návrh a implementace</b>	<b>31</b>
3.1	Úpravy podle požadavků frontendového týmu . . . . .	31
3.1.1	Interval . . . . .	31
3.1.2	Alimenty . . . . .	33
3.1.2.1	Úprava entity . . . . .	33
3.1.2.2	Spravování alimentů . . . . .	34
3.1.3	Pečovatelské dny . . . . .	35
3.1.4	Oznámení . . . . .	35
3.2	Navržené změny . . . . .	37
3.2.1	Interní DSL jazyk . . . . .	37
3.2.2	Album . . . . .	37
3.2.3	Konverter . . . . .	37
3.2.4	Našeptávač pro výjimky . . . . .	38
3.2.5	Aplikační vrstva . . . . .	38
3.2.6	Doménová vrstva . . . . .	39
3.2.7	Datová vrstva . . . . .	39
3.2.8	Implementace historie změn . . . . .	39

3.3	Návrh bezpečnosti . . . . .	41
3.3.1	OAuth 2.0 . . . . .	41
3.3.2	HTTPS . . . . .	41
3.4	Profily . . . . .	41
3.5	Databáze . . . . .	43
3.6	Návrh testování . . . . .	43
3.6.1	Unit testy . . . . .	44
3.6.2	Integrační testy . . . . .	44
3.7	Implementace bezpečnosti . . . . .	44
<b>4</b>	<b>Testování</b>	<b>45</b>
4.1	Tagy . . . . .	45
4.2	Zobrazování testů . . . . .	48
4.3	Unit testy . . . . .	49
4.4	Integrační testy . . . . .	49
4.5	Samostatný profil pro testování . . . . .	50
4.6	Pokrytí kódu testy . . . . .	51
4.6.1	JaCoCo . . . . .	51
4.6.2	IntelliJ IDEA . . . . .	51
<b>5</b>	<b>Zhodnocení a budoucí kroky</b>	<b>53</b>
5.1	Zhodnocení výsledné aplikace . . . . .	53
5.1.1	Implementace . . . . .	53
5.1.2	Testování . . . . .	54
5.1.3	Bezpečnost . . . . .	54
5.2	Požadavky na změny . . . . .	55
5.3	Návrh budoucích kroků . . . . .	55
5.3.1	Testování . . . . .	55
5.3.2	Implementace . . . . .	55
5.3.3	Bezpečnost . . . . .	55
	<b>Závěr</b>	<b>57</b>
	<b>Seznam použité literatury</b>	<b>59</b>
	<b>Seznam doporučené literatury</b>	<b>63</b>
	<b>A Seznam použitých zkratk</b>	<b>65</b>
	<b>B Slovník pojmů</b>	<b>67</b>
	<b>C Doménový model před úpravami</b>	<b>69</b>
	<b>D Testování</b>	<b>71</b>

<b>E</b>	<b>Výsledný doménový model</b>	<b>77</b>
<b>F</b>	<b>Konfigurace výsledného našeptávače pro chyby</b>	<b>79</b>
<b>G</b>	<b>Obrázky</b>	<b>81</b>
<b>H</b>	<b>Obsah přiloženého nosiče</b>	<b>89</b>

---

## Seznam obrázků

1.1	Formát pro <code>commit</code> podle pravidel definovaných pro framework Angular . . . . .	4
1.2	Ukázka použití konvence pro práci z Git . . . . .	4
1.3	Příklad definování profilu aplikace pomocí proměnné prostředí . . . . .	7
1.4	Příklad aspektově orientovaného programování v Spring . . . . .	7
2.1	Předešlý návrh kalendáře . . . . .	16
2.2	Návrh entity <code>AlimonySettings</code> . . . . .	16
2.3	Návrh entity <code>Alimony</code> . . . . .	17
2.4	Návrh <code>Need</code> . . . . .	18
2.5	Návrh entity <code>Bill</code> . . . . .	19
2.6	Návrh abstraktní entity <code>Request</code> . . . . .	20
2.7	Počet řádků kódu před začátkem práce . . . . .	21
2.8	Ukázka nastavení frameworku Swagger . . . . .	23
2.9	Návrh entit <code>Alimony</code> a <code>AlimonySettings</code> . . . . .	26
2.10	Předešlý návrh pečovatelských dnů . . . . .	27
2.11	Ukázka předešlého testování . . . . .	29
3.1	Nový návrh entity <code>Interval</code> . . . . .	32
3.2	Návrh entity <code>CareDayInterval</code> . . . . .	32
3.3	Ukázka metody vytvářející instance alimentů . . . . .	33
3.4	Ukázka konfigurace <code>cron</code> výrazu pro plánování vytváření alimentů . . . . .	34
3.5	Ukázka proměnné prostředí zapínající <code>AlimonyFactory</code> . . . . .	34
3.6	Nový návrh oznámení . . . . .	36
3.7	Ukázka instance entity <code>Interval</code> s pravidlem opakování . . . . .	37
3.8	Ukázka rozhraní <code>InterfaceConverter</code> . . . . .	38
4.1	Příklad tagu frameworku JUnit 5 . . . . .	45
4.2	Příklad třídy <code>Annotation</code> zaměňující tag s textem „ <code>integration_test</code> “ . . . . .	46
4.3	Úkoly pro spouštění testů . . . . .	47

4.4	Srovnání zobrazování kódu . . . . .	48
4.5	Příklad použití nástroje MockMvc . . . . .	50
5.1	Analýza kódu implementace . . . . .	54
5.2	Analýza kódu testů . . . . .	54
C.1	Doménový model před úpravami . . . . .	70
D.1	Seznam tříd obsahujících unit testy . . . . .	72
D.2	Seznam tříd obsahujících integrační testy . . . . .	73
D.3	Pokrytí kódů testy podle JaCoCo . . . . .	74
D.4	Pokrytí kódů testy podle JaCoCo . . . . .	75
E.1	Výsledný doménový model . . . . .	78
F.1	Konfigurace výsledného našeptávače pro řadiče . . . . .	80
G.1	Předešlý návrh entity <code>Interval</code> . . . . .	82
G.2	Předešlý návrh oznámení . . . . .	83
G.3	Návrh <code>Role</code> . . . . .	84
G.4	Nový návrh pečovatelských dnů . . . . .	85
G.5	Návrh entity <code>NeedPermissions</code> . . . . .	86
G.6	Předešlý návrh entity <code>History</code> . . . . .	87
G.7	Návrh entity <code>History</code> po změnách návrhu . . . . .	88



---

# Úvod

Rozvodové řízení je komplikovaný a nepříjemný proces. V současné době existují různé postupy, které by měly tento proces usnadnit: smluvený nesporný rozvod, předmanželská smlouva a další. Čím méně komunikace vyžaduje rozvodové řízení, tím snadnější je tento proces pro manžele. Ale v případě, že pár má děti, komunikace mezi rodiči musí pokračovat.

Jedním ze způsobů, jak ochránit děti před konfliktem samotným, je zabezpečit komunikaci rodičů pomocí aplikace. Funkce aplikace by měly pokrývat nejdůležitější aspekty, které mají vliv na děti, což je správa pečovatelských dnů, alimentů či požadavků dítěte. Takový způsob komunikace rodičů by měl, nejenom odstínit děti od konfliktu, ale i zabránit rodičům využít je v konfliktech mezi sebou.

Výše zmíněná aplikace se podle požadavků zákazníka skládá z Android aplikace, kterou současně řeší kolega Martin Beran, a serverového backendu, který je předmětem této bakalářské práce. Předěšlý stav projektu je výsledkem předmětu BI-SP1 a BI-SP2, vyučovaných na FIT ČVUT v Praze. Zmíněné předměty jsou zaměřené na studium pomocí praktického vyzkoušení analýzy, návrhu a realizace rozsáhlejšího softwarového systému. V rámci těchto předmětů byly navrženy backendové a frontendové části aplikace a byla provedená částečná implementace. Výsledná implementace serverového backendu bude poskytovat RESTové služby pro Android aplikaci a spravovat procesy, které jsou nezávislé na frontendové části aplikace. Pro implementaci byl zvolen programovací jazyk Kotlin a framework Spring.

Autor této práce se zúčastnil zmíněných předmětů a je seznámený s předešlým návrhem aplikace. Během předmětu BI-SP2 pracoval na backendu aplikace a také vystupoval v roli vedoucího backendového týmu. Během procesu implementace projektu obdržel nabídku pokračovat na backendu dané aplikace

## ÚVOD

---

v rámci bakalářské práce. Práce na projektu v rámci bakalářské práce začala ihned po dokončení předmětu BI-SP2, který byl absolvován autorem v zimním semestru akademického roku 2019/2020.

Cíle této bakalářské práce jsou navrhnout vhodné úpravy a následně implementovat serverový backend aplikace na základě existujícího návrhu a částí implementace v souladu se současným stavem implementace Android aplikace. Po dosažení funkčního výsledku zhodnotit použitelnost a navrhnout budoucí kroky.

Práce se skládá z pěti kapitol. První kapitola je věnována popisu používaných nástrojů. Druhá kapitola se zabývá analýzou současného návrhu a existujících částí implementace. Třetí kapitola představuje návrh změn a následné implementace navržených změn a ostatních funkcí. Čtvrtá kapitola je věnována testování implementovaného softwaru. V páté kapitole bude zhodnocena výsledná implementace a navrženy budoucí kroky.

---

# Teoretická část

Tato kapitola obsahuje základní teoretickou informaci o technologiích a nástrojích, které jsou použity v předešlé implementaci aplikace a také o technologiích a nástrojích, které budou použity při implementaci a analýze.

## 1.1 Pravidla frameworku Angular pro Git

V této bakalářské práci nebude použit framework Angular. Prozkoumaná pravidla nebyla kompletně převzata, ale byla provedena analýza a následně byly převzaté části pravidel, které jsou použitelné při implementaci serveru, který je předmětem této bakalářské práce. Kompletní přehled pravidel se nachází na GitHub<sup>1</sup>[1].

Pravidla, která byla použita v této bakalářské práci, definují formátování pro `commit`<sup>2</sup> v rámci verzovacího systému Git<sup>3</sup>. Příčinou zavedení konvencí je potřeba zpřehlednění grafu větví. Po dokončení této bakalářské práce, bude server kompletně funkční, ale proces vývoje tím neskončí. Budoucí kroky budou podrobně popsány v kapitole 5. Pochopení jednotlivých změn provedených během vývoje softwaru, pomáhají srozumitelné popisy jednotlivých změn. Dosažení takového výsledku pomáhá zavedení jednotného formátu pro každý `commit` v rámci projektu.

Po provedení analýzy pravidel, pravidla byly opraveny podle potřeb tohoto projektu, ale struktura nebyla změněna. Konvence byla zavedena na začátku práce autora na implementaci této bakalářské práce a neměnila se během vývoje. Na obrázku 1.2 je zobrazen kus grafu větví GitLab po zavedení pravidel.

---

<sup>1</sup>GitHub je webová platforma pro vývoj softwaru pomocí systému řízení verzí Git.

<sup>2</sup>Proces, při kterém se uloží všechny provedené změny v rámci systému řízení verzí a zařadí se do historie změn.

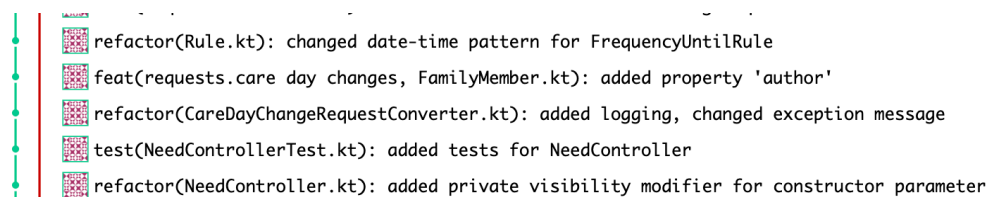
<sup>3</sup>je distribuovaný systém řízení verzí

## 1. TEORETICKÁ ČÁST

---

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

Obrázek 1.1: Formát pro `commit` podle pravidel definovaných pro framework Angular



```
refactor(Rule.kt): changed date-time pattern for FrequencyUntilRule
feat(requests.care day changes, FamilyMember.kt): added property 'author'
refactor(CareDayChangeRequestConverter.kt): added logging, changed exception message
test(NeedControllerTest.kt): added tests for NeedController
refactor(NeedController.kt): added private visibility modifier for constructor parameter
```

Obrázek 1.2: Ukázka použití konvencí pro práci z Git na grafu větví

V následujících podsekcích bude uveden podrobný popis výsledných pravidel po úpravách autora. Struktura pro `commit`, která je definovaná ve zdroji, se skládá ze pěti částí (viz obrázek 1.1):

- `typ`;
- `rozsah`;
- `předmět`;
- `tělo`;
- `zápatí`.

### 1.1.1 Typ

Typ definuje část aplikace, kterou provedený `commit` mění. Seznám hodnot, které je možné zadat, je omezený předem definovaným seznamem:

- `build` – změny procesu sestavení aplikace nebo úpravy externích závislostí;
- `ci` – změny týkající se průběžné integrace;
- `docs` – změny týkající se jenom dokumentace;
- `feta` – přidání nových funkcí;

- `fix` – oprava chyb;
- `perf` – změny zvyšující efektivitu;
- `refactor` – změny, které nepřidávají nové funkce a současně neopravují chyby;
- `style` – změny, které nemění význam kódu;
- `test` – přidání nebo úprava testů.

### 1.1.2 Rozsah

Rozsah definuje konkrétní soubory nebo balíčky, které byly změněny. Uvedený rozsah není povinný, ale v rámci této bakalářské práce bude rozsah uveden pro každý `commit`. V případě, že změny byly provedeny současně v několika různých souborech nebo balíčcích, je potřeba uvést je jako seznam s položkami v kulatých závorkách oddělenými čárkami.

### 1.1.3 Předmět

Předmět je stručným popisem provedených změn. Změny mají být odděleny od typu dvojtečkou. V případě, že `commit` obsahuje několik změn, je potřeba je oddělit čárkou. První písmeno každé změny musí být malé. Poslední změna nekončí tečkou.

### 1.1.4 Tělo

Tělo je určeno pro podrobnější popis provedených změn. Tato část není povinná, ale je doporučena v případě, že provedené změny nejsou pochopitelné po přečtení předmětu nebo není jasná příčina provedené změny.

### 1.1.5 Zápatí

Zápatí je určeno pro definování přelomových změn a definování závislostí na konkrétní požadavky v rámci GitHub nebo GitLab<sup>4</sup>. V rámci této bakalářské práce jsou každému požadavku v rámci GitLab přiděleny jednotlivé větve, proto není potřeba uvádět identifikátor požadavku v zápatí.

## 1.2 Použitý jazyk programování

Současný návrh aplikace byl proveden v jazyce Kotlin, který je relativně novým jazykem pro JVM<sup>5</sup>. Poprvé byl tento jazyk představen společností JetBrains v roce 2011. V této bakalářské práci je použita verze 1.3.72 . V porov-

---

<sup>4</sup>Webová platforma pro vývoj softwaru pomocí systému řízení verzí Git.

<sup>5</sup>Java Virtual Machine.

nání s Javou, jejíž podpora je dlouhodobě zaručena společností Oracle, nemá Kotlin předem definovanou periodu podpory, ale podle oficiálních webových stránek zaručuje zpětnou kompatibilitu pro alespoň jednu stabilní verzi, což dovoluje pohodlně migrovat do novější verze jazyka.[2][3]

Kotlin byl vytvořen jako alternativa k jazyku Java a řeší některé jeho problémy. Například, Kotlin řeší problém rozšířený použitím `null`, také známý jako *The Billion Dollar Mistake*, a problémy s ním spojené. Java samotná nemá podporu pro `not-null` proměnné, ale Kotlin takovou podporu má nativně, a to v podobě oddělení `nullable` typu pomocí `?` operátoru. Pro kompletní seznámení s jazykem Kotlin je doporučeno navštívit oficiální webové stránky, kde najdeme podrobný návod k použití jazyka.[4]

### 1.3 Nástroj pro automatizaci sestavování programu

Pro automatizaci sestavování programu se používá nástroj Gradle, který už se používá v současné implementaci aplikace. Existuje populární alternativa – Maven, ale Gradle je modernější a pohodlnější. Konfigurace se provádí v jazyce Groove nebo přímo v jazyce Kotlin. Také Gradle poskytuje možnost definovat vlastní úkoly spustitelné pomocí příkazové řádky, které pak budou využité pro analýzu a spuštění testů. Podrobnější informaci o porovnání nástrojů Gradle a Maven lze najít na webových stránkách [5] a [6].

### 1.4 Nástroj pro vývoj podnikových aplikací

Spring je jeden z nejznámějších frameworků určených pro vývoj podnikových aplikací. Infrastruktura Spring je velká a komplikovaná. Aktuálně se Spring skládá ze 24 aktivních projektů. V rámci této bakalářské práce bude využito současně několik různých projektů, které zajišťují korektní funkčnost různých aspektů serveru. V této sekci budou popsány jenom projekty, které budou následně využity při implementaci. Popis je velice stručný a je uveden za účelem pochopení základů použitých technologií. Pro podrobnější informace o zmíněných projektech je doporučeno navštívit oficiální webovou stránku [7].

#### 1.4.1 Spring framework

Spring framework je základem pro ekosystém různých projektů, které budou popsány v následujících sekcích.[8] Tento nástroj samotný je správcem závislostí (*dependency manager*). Spravování závislostí je reprezentováno pomocí principu *Inversion of Control* (IoC), což znamená, že framework Spring je kontejnerem, který je zodpovědný za vytváření objektů a jejich provázáním mezi sebou. Konfigurace aplikace je reprezentována pomocí rozhraní `ApplicationContext`.

```
spring.profiles.active=dev
```

Obrázek 1.3: Příklad definování profilu aplikace pomocí proměnné prostředí

```
@Bean
@ConditionalOnProperty(
    value = ["scheduled.alimonyFactory"],
    matchIfMissing = false,
    havingValue = "true")
fun startAlimonyFactory(): AlimonyFactory {
    logger.info("Schedule job: starting AlimonyFactory")
    return AlimonyFactory(
        alimonySettingRepository,
        alimonyRepository)
}
```

Obrázek 1.4: Příklad aspektově orientovaného programování v Spring

Pro pohodlnou práci a zvýšení efektivity napsaného kódu má Spring množství dalších užitečných řešení. Jak už bylo zmíněno, popisovat všechno nemá smysl, ale několik nejdůležitějších věcí není možné opomenout. První funkce, která bude široce využita při implementaci, se týká proměnných v rámci prostředí aplikace. Tyto proměnné umožňují definovat důležité proměnné pro konfiguraci běhu aplikace zvláště od jejich využití v kódu. Framework Spring umožňuje využít tyto proměnné přímo v kódu. Implicitním souborem obsahujícím tyto proměnné je soubor `application.properties`. Pro zavedení různých proměnných pro různé případy použití aplikace je možné definovat několik takových souborů pro každý z profilů. Jedinou věcí, kterou je potřeba udělat, je vytvořit soubor, který bude mít v názvu jméno profilu, kterému patří (`application-profile.properties`). Pro definování, který profil má aplikace použít v následujících spouštěních, je potřeba v implicitním souboru zadat potřebný profil jako proměnnou (viz obrázek 1.3).

Spring framework používá techniku vkládání závislosti (*dependency injection*), což dává možnost psaní kvalitnějšího softwaru, ale Spring poskytuje možnost zlepšit výsledný kód ještě více. Framework Spring také používá paradigma aspektově orientovaného programování (AOP). Toto paradigma má za cíl zvýšit modularitu výsledného softwaru pomocí rozdělení kódu do logických částí, nalezení opakujících se částí, které jsou také nazývané průřezové problémy, a nahrazení opakujícího se kódu. Příkladem AOP je anotace zaměřená na podmíněné spouštění metody v závislosti na výsledku předem definované podmínce v závorkách (viz obrázek 1.4).

### 1.4.2 Spring Boot

Spring Boot je pomocný nástroj pro programátora při konfiguraci aplikace využívající framework Spring. Spring Boot registruje 17 různých implicitních zdrojů proměnných. Kompletní popis zdrojů je na oficiálních stránkách dokumentace [9]. Podle nalezených proměnných framework provádí analýzu konfigurace aplikace a následně, pomocí anotací pro podmínky, automaticky nastavuje prostředí aplikace. Navíc od klasického frameworku Spring, Spring Boot obsahuje rozšířený seznam anotací pro podmínky, které jsou široce využity pro zmíněnou automatickou konfiguraci.[10]

### 1.4.3 Spring Security

Spring Security je framework určený pro implementaci autentizace pro identifikaci uživatele a autorizace pro zaručení přístupu do jednotlivých aspektů aplikace. Framework vyžaduje využití frameworku Spring, případně i frameworku Spring Boot. Podrobné informace lze najít na oficiálních stránkách [11].

### 1.4.4 Spring Data

Spring Data je framework pro pohodlnou práci programátora při implementaci datové vrstvy<sup>6</sup>. Framework vytváří dodatečnou vrstvu nad implementací Java Persistence API<sup>7</sup> (JPA). Implementací JPA může být EclipseLink nebo Hibernate. Příkladem využití je rozhraní `CrudRepository`, které bude využito v této bakalářské práci. Rozhraní definuje základní operace s databází a dovoluje programátorovi jenom definovat typ entity a typ primárního klíče. V případě, že programátor potřebuje složitější dotazy na databázi, framework poskytuje možnost definování metod, které ve svých názvech obsahují popis dotazů [12]. Framework rozebírá název na jednotlivé části a překládá ho na SQL<sup>8</sup> dotaz.

### 1.4.5 Spring Web Services

Spring Web Services je framework určený pro implementaci webových služeb. V rámci této bakalářské práce bude tento framework využit pro implementaci aplikační vrstvy serveru. Aplikační vrstva aplikace má pokrývat případy užití aplikace pomocí příslušných řadičů. Podrobnou informaci lze najít na oficiálních stránkách [13].

---

<sup>6</sup>Vrstva aplikace, která komunikuje s databází.

<sup>7</sup>Application programming interface.

<sup>8</sup>Structured Query Language.



## 1.5 Testování

Testování aplikace se skládá z několika částí. První část je testování softwaru samotného pro ověření jestli jednotlivé metody, třídy nebo celé moduly fungují správně a komunikují mezi sebou bezchybně. Druhou částí testování je ověření, jestli testy pokrývají dostatečnou část kódu a jestli pokrývají důležité funkce. V této sekci jsou popsány nástroje pro zaručení kvalitního testování, které budou následně využity při implementaci testů a jejich analýze.

### 1.5.1 Spring

Framework Spring poskytuje velký počet nástrojů pro testování softwaru. Každý z dodatečných frameworků, jako Spring Boot nebo Spring Security, spolu s novými funkcemi, přidávají vhodné nástroje pro kvalitní testování těchto funkcí. Celý přehled nástrojů, které poskytuje framework, je na oficiálních stránkách dokumentace [14]. V této sekci jsou uvedeny jenom nejdůležitější použité nástroje.

Prvním nástrojem je anotace `SpringBootTest`, kterou poskytuje framework Spring Boot pro implementaci integračních testů. Při použití této anotace framework vytvoří nutný kontext aplikace pro kompletní testování.

Druhým důležitým nástrojem je třída `MockMvc`, která poskytuje možnost kvalitního testování aplikační vrstvy aplikace. Tento nástroj je součástí frameworku Spring Web Services.

### 1.5.2 JUnit

JUnit je framework určený pro testování softwaru. V této bakalářské práci bude použita verze 5. Tento framework bude použit pro vytvoření kostry testování. Podrobnější informace o funkcích frameworku je dostupná na oficiálních webových stránkách [15].

### 1.5.3 JaCoCo

JaCoCo je nástroj poskytující podrobnou analýzu testů. Tento nástroj bude použit pro výslednou analýzu implementovaných testů a také jako pomocný nástroj pro nalezení důležitých neotestovaných funkcí během implementace testů. Nástroj existuje v reprezentaci přídatného modulu pro nástroj Gradle, který se již používá v současné implementace programu. Výsledky práce jsou následně uloženy do předem definované složky a jsou reprezentovány pomocí HTML<sup>9</sup> stránek. Nástroj je otevřeným softwarem, proto podrobnou informaci o implementaci lze najít na oficiálních stránkách [16]. Výsledky analýzy podle JaCoCo jsou dostupné v příloze D.

---

<sup>9</sup>Hypertext Markup Language.

### 1.5.4 IntelliJ IDEA

Pro provedení analýzy testů také bylo využito vývojové prostředí IntelliJ IDEA, používané autorem této práce pro implementaci programu. Nástroj pro pokrytí kódu testy je implicitním nástrojem zmíněného vývojového prostředí a poskytuje možnost zobrazování výsledků analýzy přímo v kódu. Takový přístup zrychluje proces implementace testů. Nástroj také dovoluje vygenerovat výsledky zvlášť od kódu, proto tyto výsledky jsou také dostupné v příloze D.

## 1.6 Dokumentace

Pro dokumentace API v současné implementaci aplikace se používá framework Swagger. Důležitou funkcí, kterou framework poskytuje, je automatické generování dokumentace na základě přidaných anotací v kódu. Dokumentace je následně vygenerována automaticky a je dostupná z internetu ze stejné adresy a portu jako server. Podrobná informace o frameworku je dostupná na oficiálních webových stránkách [17].

## 1.7 Použité databáze

### 1.7.1 H2

Současná implementace používá relační databázi H2. Hlavním důvodem zvolení právě této databáze je možnost volby několika různých režimů, kde jeden z režimů je vestavěný režim. Tento režim usnadňuje programátorovi práci s databází, protože se databáze vytváří při spuštění programů a zaniká při jejich zastavení. Soubory s daty jsou uloženy na disku nebo přímo v paměti. H2 má též vlastního manažera, který je dostupný přes webový prohlížeč. Podrobnější informace jsou dostupné na oficiálních webových stránkách [18].

### 1.7.2 PostgreSQL

PostgreSQL je široce využívanou objektově relační databází napsanou v jazyce C. V době odevzdání této bakalářské práce je databáze PostgreSQL udržována skupinou Global Development Group a je otevřeným softwarem. Pro implementaci byla zvolena poslední stabilní verze – 12. Kompletní dokumentace je dostupná online [19]. Framework Spring, zmíněný v sekci 1.4, nativně podporuje použití PostgreSQL.

## 1.8 Nástroj pro analýzu rozsahu implementace

Nástroj CLOC<sup>10</sup>[20] poskytuje možnost spočítat počet řádek kódu je v dané složce. Nástroj podporuje velký počet programovacích jazyků. Výsledek obsahuje počet řádek kódu oddělený od komentářů a prázdných řádek. Tento nástroj bude použit pro analýzu současné implementace aplikace a analýzu provedené práce v rámci bakalářské práce.

---

<sup>10</sup>Count Lines of Code.



---

# Analýza

Tato kapitola se zabývá analýzou již existujícího návrhu a dosavadního stavu implementace. Pro zajištění verze aplikace, která bude předmětem analýzy této kapitoly, uvádím datum ukončení práce na projektu v rámci předmětu Softwarový týmový projekt 2 (BI-SP2) – 25. února 2020. Také pro možnost pohodlného vyhledání konkrétní verze (<https://gitlab.fit.cvut.cz/rozvody/be-springboot>), uvádím poslední commit provedený přes distribuovaný systém řízení verzí – Git – v rámci tohoto předmětu:

„252b0288dbfe9942446b78fd452c0edce810a370“

## 2.1 Předmět Softwarový týmový projekt 1

V rámci předmětu Softwarový týmový projekt 1 (BI-SP1) pracovalo sedm lidí včetně autora této práce. Tým měl za úkol analyzovat požadavky zákazníka a navrhnout implementaci aplikace. Během semestru tým provedl kompletní analýzu požadavků zákazníka. Tým navrhl scénáře užití aplikace:

- Přihlašování/Registrace;
- Přihlašování/Registrace do rodiny;
- Role v aplikaci a jejich vytváření;
- Nastavení pečovatelských dnů;
- Kalendář;
- Kniha potřeb dítěte;
- Uchování účtenek;
- Správa alimentů.

Potom byl navržen diagram užití<sup>11</sup> a diagram aktivit<sup>12</sup>, podle kterých byl navržen drátový model pro frontendovou část aplikace a doménový model<sup>13</sup> pro backendovou část aplikace.

Výsledný návrh aplikace se skládá ze dvou částí. Frontendová část aplikace je reprezentována Android aplikací, kterou současně řeší kolega Martin Beran v rámci své bakalářské práce. Backendová část aplikace, která je předmětem této bakalářské práce, je reprezentována serverovým backendem poskytující REST API pro Android aplikaci.

### 2.2 Předmět Softwarový týmový projekt 2

Cílem předmětu Softwarový týmový projekt 2 (BI-SP2) byla implementace návrhu předmětu BI-SP1. Autor této práce pracoval v backendovém týmu a zároveň vystupoval v roli vedoucího backendového týmu.

Pro vývoj backendové části aplikace byl zvolen jazyk Kotlin, zmíněný v sekci 1.2, a framework Spring, zmíněný v sekci 1.4. Jako nástroj pro automatizaci sestavování programu byl zvolen nástroj Gradle. Podrobněji budou výsledky předmětu BI-SP2 probrány v sekci 2.5.

### 2.3 Doménový model

Hlavním zdrojem informace o výsledném návrhu serveru je doménový model. Kompletní doménový model se nachází v příloze C. Během implementace tým označoval entity, které jsou kompletně nebo částečně implementovány. Zelenou barvou jsou označeny třídy, které už jsou implementovány. Žlutou barvou jsou označeny třídy, které ještě nejsou implementovány. Třídy označené zároveň žlutě i zeleně jsou implementovány pouze částečně. Tento doménový model má nedostatky detekované při implementaci, jak frontendové, tak i backendové části aplikace. Jako příklad takových nedostatků je možné uvést zbytečně komplikovaný návrh entity `Interval` (viz obrázek G.1). Podrobný popis tohoto problému bude uveden v sekci 2.6.

### 2.4 Analýza předešlého návrhu

V této sekci budou podrobně popsány klíčové aspekty aplikace podle již existujícího návrhu aplikace. Problémy a navržené změny těchto částí budou popsány v následujících sekcích.

---

<sup>11</sup>Popisuje chování systému z vnějšího pohledu.

<sup>12</sup>Zobrazuje jak objekty spolupracují.

<sup>13</sup>Náčrt základních entit systému a vztahů mezi nimi.

### 2.4.1 Registrace a přihlášení do systému

Aplikace je navržena takovým způsobem, že první uživatel, který má vytvořit svůj účet je jeden z rodičů. Pro registraci je potřeba řada povinných údajů:

- jméno – zvolené jméno se stává implicitním jménem v systému;
- příjmení – zvolené příjmení se stává implicitním příjmením v systému;
- e-mailová adresa – zvolený e-mail je identifikátorem uživatele v rámci systému;
- heslo – heslo pro autorizaci v systému.

Na základě těchto údajů se vytvoří unikátní uživatel v rámci systému. V tento okamžik uživatel není přihlášen do žádné rodiny a nemá žádnou roli. Podrobněji budou role popsány v sekci 2.7.1.

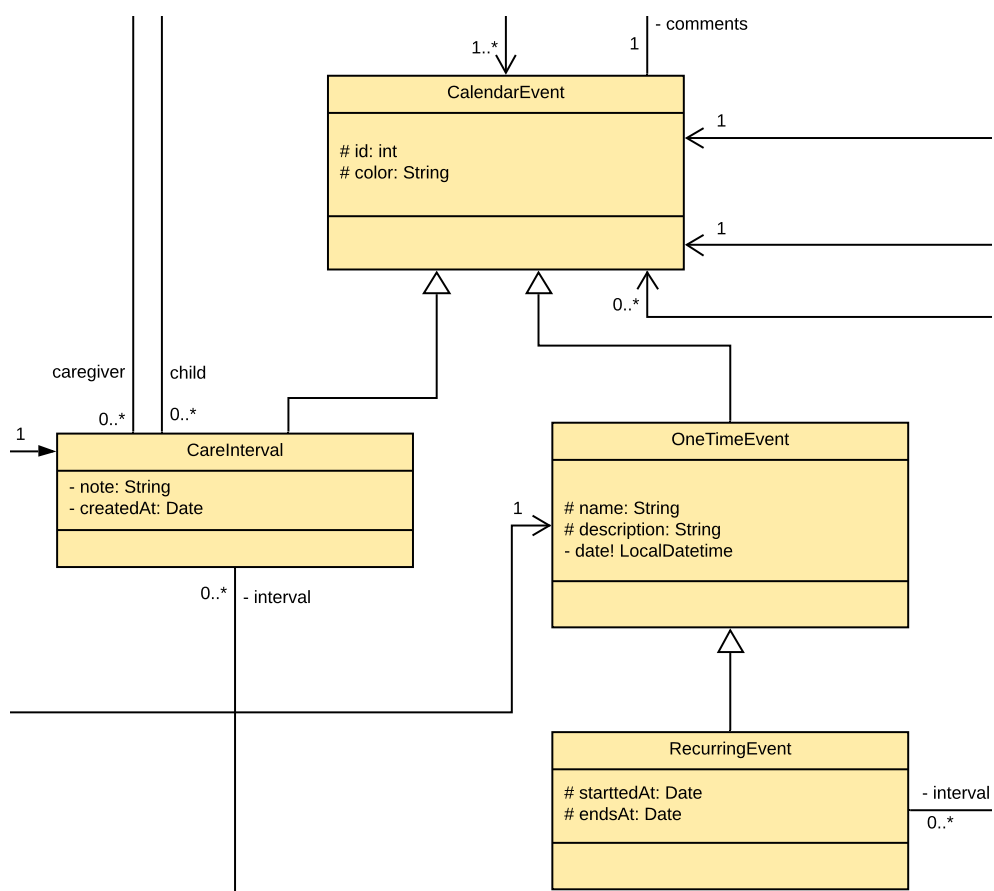
Potom má uživatel možnost vytvořit rodinu nebo se přihlásit do již existující rodiny. Pro vytvoření nové rodiny, potřebuje uživatel zadat jméno rodiny a přidat členy rodiny. Autor této rodiny se automaticky stává jedním z rodičů této rodiny. Jinou možností je se přihlásit do rodiny, která již existuje. Podmínkou je pozvání do některé existující rodiny, což znamená, že přidat nového uživatele do rodiny může jenom člen této rodiny. V takovém případě uživatel už nemá možnost zvolit role v rámci rodiny. Role má být nastavena uživatelem, který vytvořil toto pozvání.

### 2.4.2 Kalendář

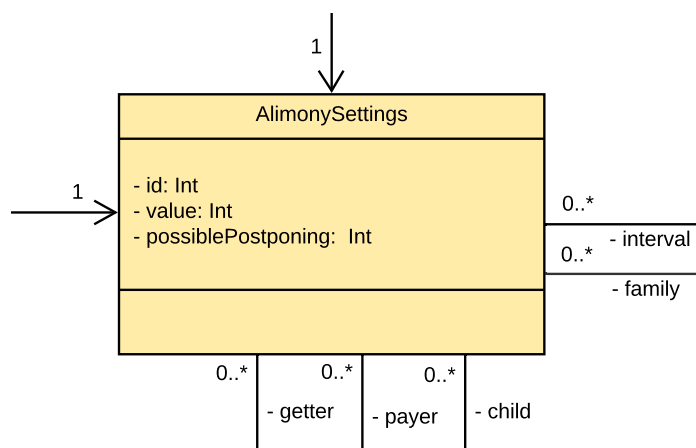
Kalendář je nejdůležitějším zdrojem informací pro celou rodinu a je společný pro všechny uživatele. Na něm jsou zobrazené pečovatelské dny obou rodičů zvýrazněné různými barvami. Kalendář též zobrazuje jednorázové a pravidelné události.

Doménový model neobsahuje informace o kalendáři, ale obsahuje entity reprezentující informaci, kterou kalendář bude zobrazovat (viz obrázek 2.1). Každý element, který bude zobrazen v kalendáři, je reprezentován stejnou entitou **CalendarEvent**. Entita obsahuje jenom informaci o barvě, kterou událost bude zobrazená v kalendáři a závislost na entitě **Comment**. Za reprezentaci pečovatelských dnů odpovídá entita **CareInterval**, která se dědí od entity **CalendarEvent**. Za reprezentaci jednorázových a opakujících se událostí odpovídají entity **OneTimeEvent** a **RecurringEvent**, kde se **RecurringEvent** dědí od **OneTimeEvent**.

## 2. ANALÝZA

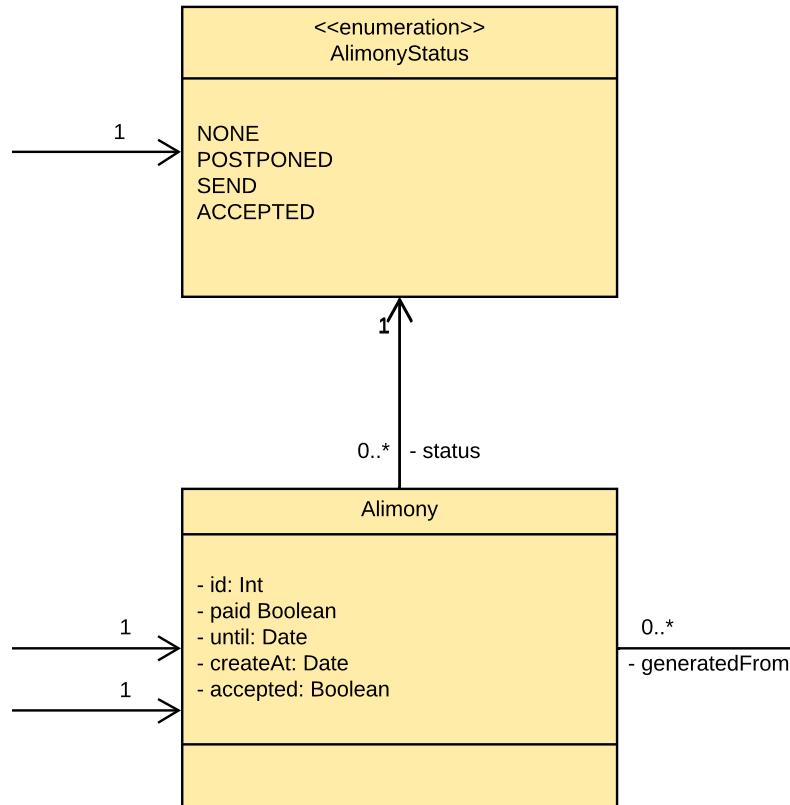


Obrázek 2.1: Návrh kalendáře podle doménového modelu z předmětu BI-SP2



Obrázek 2.2: Návrh entity AlimonySettings podle doménového modelu z předmětu BI-SP2



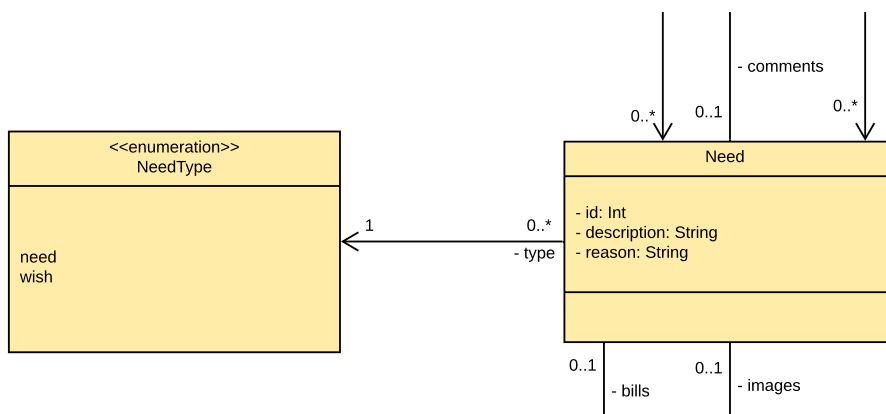


Obrázek 2.3: Návrh entity Alimony podle doménového modelu z předmětu BI-SP2

### 2.4.3 Alimenty

Důležitou částí aplikace je správa alimentů, které má pravidelně uhrazovat jeden z rodičů. Tento proces byl rozdělen do dvou částí. První částí je dlouhodobé nastavení alimentů (viz obrázek 2.2). Druhou částí jsou samotné alimenty (viz obrázek 2.3), které se generují na základě dlouhodobých nastavení. Jedna rodina může mít zároveň několik nastavení v případě, že rodina má několik dětí nebo chce rozdělit alimenty do logických bloků.

Každá instance alimentů má stav, který se postupně mění. V moment vytvoření instance má stav **NONE**. Po odeslání alimentů druhému rodiči se stav mění na **SEND**. Druhý rodič následně potvrdí, že alimenty přijal a tím se změní stav na **ACCEPTED**. Také může nastat situace, kdy rodič nemá možnost odeslat alimenty včas. V takovém případě se stav instance mění na **POSTPONED**.



Obrázek 2.4: Návrh entity Need podle doménového modelu z předmětu BI-SP2

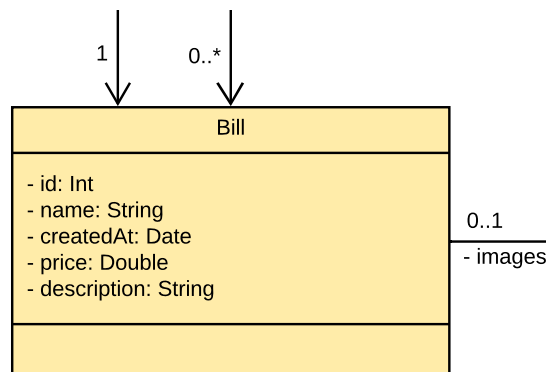
#### 2.4.4 Kniha potřeb dítěte

Jedním s častých problémů, které vznikají během procesu rozvodu, je nakupování příliš drahých dárků, o kterých ostatní členové rodiny nevědí. Jako příklad je možné uvést nakupování bot pro dítě. Jeden z rodičů si může chtít „koupit lásku dítěte“ a koupí několikanásobně dražší boty než, dítě ve skutečnosti potřebuje. Kniha potřeb dítěte (viz obrázek 2.4) je zaměřena na překonání takových situací.

Potřeba může být typu *need* nebo *wish*. Podle dosavadního návrhu je rozdíl mezi typy pouze pro informační účely. Každé instanci entity *Need* patří instance entity *NeedPermission* (viz obrázek G.5), která definuje přístupová práva pro jednotlivé členy rodiny. V případě, že uživatel nemá žádné z přístupových práv, potřeba se nevyskytuje v jeho seznamu potřeb dítěte. Toto pravidlo se netýká jenom rodičů, kteří mají přístup ke všem potřebám automaticky.

Potřeba obsahuje následující informaci:

- popis – popis potřeby;
- příčina – příčina, proč dítě toto potřebuje;
- obrázky – obrázky věcí, které dítě potřebuje;
- účtenky – účtenky v případě, že někdo z rodičů splnil potřebu;
- komentáře – komentáře členu rodiny včetně dítěte.

Obrázek 2.5: Návrh entity `Bill` podle doménového modelu z předmětu BI-SP2

### 2.4.5 Účtenky

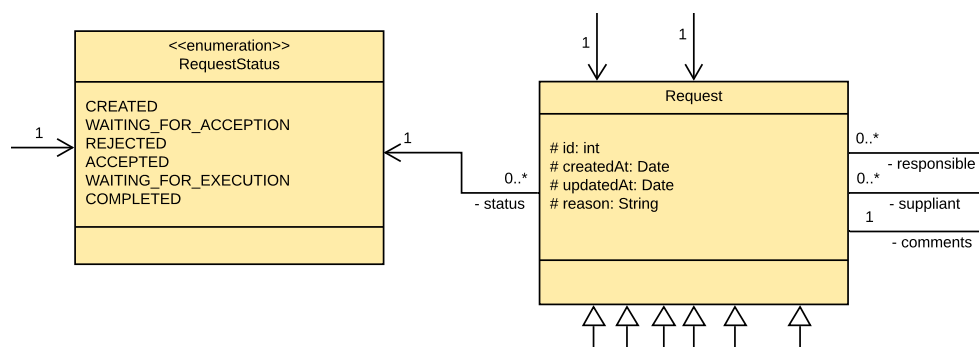
V sekci 2.4.4 už bylo zmíněno, že je potřeba řešit problém nakupování dárků pro dítě, ale kniha potřeb dítěte řeší tento problém jenom částečně. Nikdo nezabrání uživateli uvést v popisu splněného požadavku nebo potřeby neplatné údaje, proto byla zavedena možnost přidání účtenky (viz obrázek 2.5). Tato entita, kromě údajů o nakoupeném zboží, umožňuje přidání fotografií potvrzujících platnost uvedených údajů.

### 2.4.6 Požadavky na změny

Každý člen rodiny může udělat určité změny v rámci rodiny. Například nastavit přezdívku pro konkrétního uživatele nebo přidat fotografii do události v kalendáři. Každá taková změna je důležitá, ale jsou změny, které by měly být kontrolovány rodiči. Pro zaručení kontroly byly zavedeny požadavky na změny pro některé změny v rámci rodiny. Následně, pro provedení změny v příslušných entitách uživatel potřebuje nejdřív vytvořit požadavek a schválit ho. Právo na schválení nebo zamítnutí požadavků mají jenom rodiče. Požadavky, které byly vytvořeny jedním z rodičů, mají být potvrzeny druhým rodičem. Ostatní požadavky musí být potvrzeny oběma rodiči. Seznám požadavků se skládá z:

- `AlimonySettingRequest` – požadavek na změnu nastavení alimentů, který může být vytvořen jedním z rodičů;
- `AlimonyChangeRequest` – požadavek na změnu stavu instance alimentů, který může být vytvořen jedním z rodičů;
- `OneTimeEventRequest` – požadavek na vytvoření jednorázové události v kalendáři, který může být vytvořen jakýmkoliv členem rodiny;

## 2. ANALÝZA



Obrázek 2.6: Návrh abstraktní entity **Request** podle doménového modelu z předmětu BI-SP2

- **OneTimeEventChangeRequest** – požadavek na změnu již existující události v kalendáři, který může být vytvořen jakýmkoliv členem rodiny;
- **ChildItemRequest** – požadavek na změnu nastavení pečovatelských dnů, který může být vytvořen jakýmkoliv členem rodiny. Změna může být jak jednorázová, tak i dlouhodobá.

Každá entita požadavku je zděděna od společné abstraktní entity **Request** obsahující základní informace o požadavku (viz obrázek 2.6). Mezi základní informace patří datum vytvoření, datum poslední změny, příčina a také aktuální stav požadavku. Také tato entita obsahuje závislosti na základních aktérech tohoto požadavku a komentáře.

Stavy požadavků reprezentují jejich životní cyklus. Při vytvoření každý požadavek má stejný stav – **CREATED**. Po potvrzení požadavku jeho autorem má požadavek stav **WAITING\_FOR\_ACCEPTING**. Po schválení rodiči se stav mění na **ACCEPTED** nebo na **WAITING\_FOR\_EXECUTION** v případě, že je změna akceptována, ale ještě není uplatněna. Též může nastat případ, že byl požadavek zamítnut, pak se stav mění na **REJECTED**. Požadavek je zamítnut v případě, že alespoň jeden z rodičů ho zamítl. V okamžik, kdy je požadavek schválen nebo se čeká na uplatnění, není možné měnit položky tohoto záznamu.

### 2.4.7 Historie změn

Každá provedená v rámci rodiny změna má význam při debatách mezi rodiči. Podle požadavků zákazníka by všechny změny v rámci rodiny měly být zaznamenány. Proto byly zavedeny entity historií, které kopírují všechny položky entit a přidávají datum vytvoření záznamu a odkaz do příslušné entity v pří-

```

72 text files.
72 unique files.
2 files ignored.

github.com/AlDanial/cloc v 1.84 T=0.07 s (1015.0 files/s, 15993.1 lines/s)
-----
Language             files      blank      comment      code
-----
Kotlin                70         220          26          857
SUM:                  70         220          26          857
-----

```

Obrázek 2.7: Počet řádků kódu před začátkem práce

padě, že tato příslušná entita byla aktualizována. Takové entity jsou zavedeny jenom pro důležité části aplikace, které mohou být změněny uživatelem:

- `CalendarEventHistory`;
- `BillHistory`;
- `AlimonyHistory`;
- `AlimonySettingsHistory`;
- `RequestHistory`.

Všechny typy historie jsou zděděny od stejné entity – `History`. Tato entita obsahuje základní údaje pro libovolný typ historie: `ID`<sup>14</sup> a datum vytvoření záznamu.

## 2.5 Analýza předešlé implementace

Dosavadní implementace obsahuje 69 souborů a 845 řádek kódů (viz obrázek 2.7). Práce byla provedena členy týmu předmětu BI-SP2. Za účelem zpřesnění analýzy byl zvolen nástroj pro analýzu počtu řádek kódu, který byl popsán v sekci 1.8. Implementace částečně pokrývá doménový model zmíněný v sekci 2.3. V této sekci budou popsány již implementované části aplikace.

Pro implementaci serverového backendu byla zvolena architektura REST skládající se ze tří vrstev:

- Aplikační vrstva, která pokrývá scénáře využití aplikace;
- Doménová vrstva, která obsahuje logiku aplikace;
- Datová vrstva, která komunikuje s databází.

<sup>14</sup>identifikátor

Pro implementaci datové vrstvy byl použit framework Spring Data, který byl popsán v sekci 1.4. Nástroj přidává dodatečnou vrstvu pro implementaci JPA a tím zrychluje a zjednodušuje implementaci. Pro implementaci této vrstvy bylo zvoleno rozhraní `CrudRepository`, které vyžaduje uvedení typu doménové třídy a typu identifikátoru. Každé entitě patří jedna taková třída pro práci s databází.

Pro implementaci doménové vrstvy byly pro každou entitu přidány rozhraní reprezentující funkce, které je možné provádět s příslušnou entitou. Implementace byla oddělená od definicí metod pro zvětšení modularity výsledného softwaru. Implementace této vrstvy přímo komunikuje s datovou vrstvou. Provázání tříd se vzniká na základě vkládání závislostí. Proces vkládání řídí framework Spring podle principu IoC. Tato vrstva komunikuje s datovou vrstvou pomocí reprezentací entit, ale s aplikační vrstvou komunikuje pomocí Data Transfer Object (DTO). Konverzace z entity na DTO se provádí v příslušné třídě entity. Konverzace z DTO na entitu se provádí v příslušné třídě DTO.

Aplikační vrstva je reprezentována sadou řadičů (*controllers*), kde každý řadič je určen pro konkrétní aspekt použití aplikace. Pro každý řadič je vymezená samostatná cesta, která se zadává jako řetězec požadavku [21]. Tato vrstva přímo komunikuje s doménovou vrstvou. Provázání tříd se provádí také pomocí vkládání závislostí. Komunikace aplikační vrstvy přes API se provádí pomocí DTO.

### 2.5.1 Implementované entity

Seznam implementovaných entit:

- `AlimonyStatus`;
- `Bill`;
- `CalendarEvent`;
- `OneTimeEvent`;
- `Comment`;
- `FamilyMember` – je přítomná pouze část implementace;
- `History`;
- `IntervalType`;
- `NWeekInterval`;
- `WeekInterval`;
- `NeedAccess`;
- `NeedType`;

- `AbstractPermissions`;
- `Permissions`;
- `RequestStatus`;
- `User`.

Typ chyby	HTTP status	zpráva	URL
Illegal Access	401	původní zpráva chyby	původní cesta
Illegal Argument	400	původní zpráva chyby	původní cesta
Null Pointer	500	původní zpráva chyby	původní cesta
No Such Element	404	nic	nic

Tabulka 2.1: Ukázka konfigurace našeptávače zachycování výjimek pro řadiče

```

1  @Configuration
2  @EnableSwagger2
3  class SwaggerConfig {
4  @Bean
5  fun api(): Docket {
6      return Docket(DocumentationType.SWAGGER_2)
7          .select()
8          .apis(RequestHandlerSelectors.any())
9          .paths(PathSelectors.any())
10         .build()
11     }
12 }

```

Obrázek 2.8: Ukázka nastavení frameworku Swagger

Kromě kostry aplikace, která zaručuje třívrstvou architekturu, byly také implementovány pomocné třídy zlepšující výsledný návrh aplikace. Jednou z takových tříd je třída, která uvádí nápovědy pro všechny řadiče ohledně zachycování výjimek. Tato třída byla zavedena za účelem poskytnutí uživateli pouze korektně formátované informace a filtrování zbytečných informací pro koncového uživatele. Konfigurace našeptávače je uvedena v tabulce 2.1. Jinou pomocnou třídou je řadič určený pro ověření, zda server funguje. Tento řadič je namapován na cestu „/“.

### 2.5.2 Dokumentace API

Pro dokumentaci API byl zvolen framework Swagger, který již byl zmíněn v sekci 1.6. V předešlé implementaci je použita druhá verze tohoto frameworku.

Také byla provedena konfigurace pro generování dokumentace při každém spuštění aplikace na základě do kódu přidávaných anotací (viz obrázek 2.8). Po spuštění aplikace je online dokumentace dostupná na cestě „/swagger-ui.html“.

### 2.5.3 Profily

Framework Spring, použitý v předešle implementaci, poskytuje možnost rozdělit aplikaci do logických bloků, které budou existovat jenom v konkrétních profilech.[22] Implicitně všechny komponenty nezávisí na aktuálně zvoleném profilu. Pro zavedení profilů<sup>15</sup> pro konkrétní komponentu je potřeba ji označit anotací `@Profile`. V závorkách vedle anotací je potřeba přidat seznam profilů, ve kterých tato komponenta bude existovat. Konfigurace aktuálně zapnutých profilů se provádí pomocí souboru `application.properties`, který definuje proměnné pro prostředí aplikace. Soubor se nachází ve složce s cestou „be-springboot/src/main/resources“.

Každý profil také může obsahovat vlastní konfigurační soubor, který definuje všechny nutné proměnné prostředí. Soubor má být zadán ve formátu `application-{profile}.properties`, kde `profile` je názvem profilu kterému patří tento soubor. Dosavadní návrh aplikace obsahuje dva konfigurační soubory.

První konfigurační soubor obsahuje implicitní proměnné pro prostředí aplikace. Aktuálně má soubor jenom definici aktuálního profilu aplikace.

Druhý konfigurační soubor patří profilu `development`. Tento profil je určen pro pohodlný proces vývoje aplikace. Soubor obsahuje konfiguraci databáze a konfiguraci logování aplikace. Podrobněji bude použitá databáze popsána v sekci 2.5.4.

### 2.5.4 Databáze

Pro proces vývoje byla zvolena jednoduchá relační databáze H2, která nevyžaduje nastartování serveru zvlášť od aplikace. Tato databáze má též režim, při kterém jsou všechna data uložena přímo v paměti aplikace. Podrobněji byla tato databáze a její princip fungování popsány v sekci 1.7.

## 2.6 Analýza požadavků na změny frontendové části aplikace

V rámci předmětu BI-SP2 probíhala současně s implementací backendové částí aplikace implementace frontendové částí aplikace, která je reprezentovaná Android aplikací. Během vývoje frontendové části aplikace byly identi-

---

<sup>15</sup>Jedna komponenta může současně patřit několika různým profilům.



fikovány nedostatky, které zbytečně komplikují implementaci, jak backendu, tak i frontendu. V této sekci budou popsány jednotlivé požadavky na změny od frontendového týmu.

### 2.6.1 Interval

Prvním takovým požadavkem je změna entity `Interval` (viz obrázek G.1), která je v projektu široce využívána. Návrh řešení tohoto problému bude popsán v sekci 3.1. Zde bude popsán pouze problém samotný. Entita `Interval` reprezentuje časové rozmezí pro pečovatelské dny, opakované události, nastavení alimentů a navazující požadavky na ně na změny a historické záznamy.

Jádro problému je v tom, že entita je navržena pomocí generalizace, neboli dědičnosti z hlediska implementace. Takový návrh dává možnost vytvořit konkrétní typ intervalu pomocí zvolení odpovídající třídy. Na druhou stranu takový návrh působí komplikaci při implementaci a současně nepokrývá všechny možné případy využití intervalů. Například, není možné sestavit interval, který se bude opakovat každý poslední den měsíce. Pokud bychom se chtěli držet aktuální implementace a zároveň pokrýt všechny možné případy, ztratili bychom přehlednost zvolení správné třídy při vytváření instance.

Dalším problémem návrhu entity `Interval` je provázanost s pečovatelskými dny. Při návrhu frontendové části aplikace bylo zjištěno, že dlouhodobá nastavení pečovatelských dnů rodičů nepotřebují komplikované nastavení a zároveň by potřebovaly mít odkazy na konkrétního rodiče, který je zodpovědný za dítě v tento den nebo časový úsek. Proto je potřeba oddělit tyto intervaly pečovatelských dnů zvlášť od ostatních intervalů.

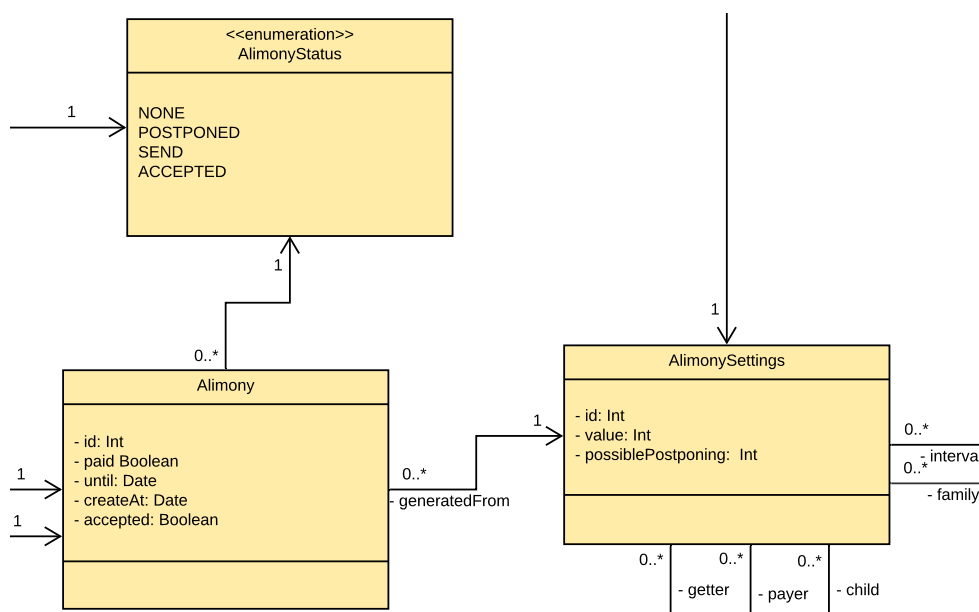
### 2.6.2 Alimenty

Entita `Alimony` reprezentuje alimenty, které jeden rodič má posílat druhému rodiči. Jedna instance odpovídá alimentům za jeden měsíc. Především návrh popisuje entitu a její navázanost na entitu `AlimonySettings` (viz obrázek 2.9). Nastavení alimentů definují dlouhodobou konfiguraci. Potom se na základě této konfigurace vytváří jednotlivé instance alimentů. Nastavení se definují v rámci jedné rodiny. V případě potřeby mají rodiče možnost rozdělit alimenty do několika nastavení, které budou současně validní.

Hlavní problém je v tom, že instance alimentů by se měly vytvářet nezávisle na frontendové části aplikace, což není popsáno v předešlém návrhu aplikace. Podrobně návrh řešení problému bude podrobně popsán v sekci 3.1.2.

Dodatečný požadavek frontendového týmu se týká entity `Alimony` samotné. Za účelem zjednodušení návrhu frontendové části aplikace je potřeba přidat závislost na entitu `Family`.

## 2. ANALÝZA

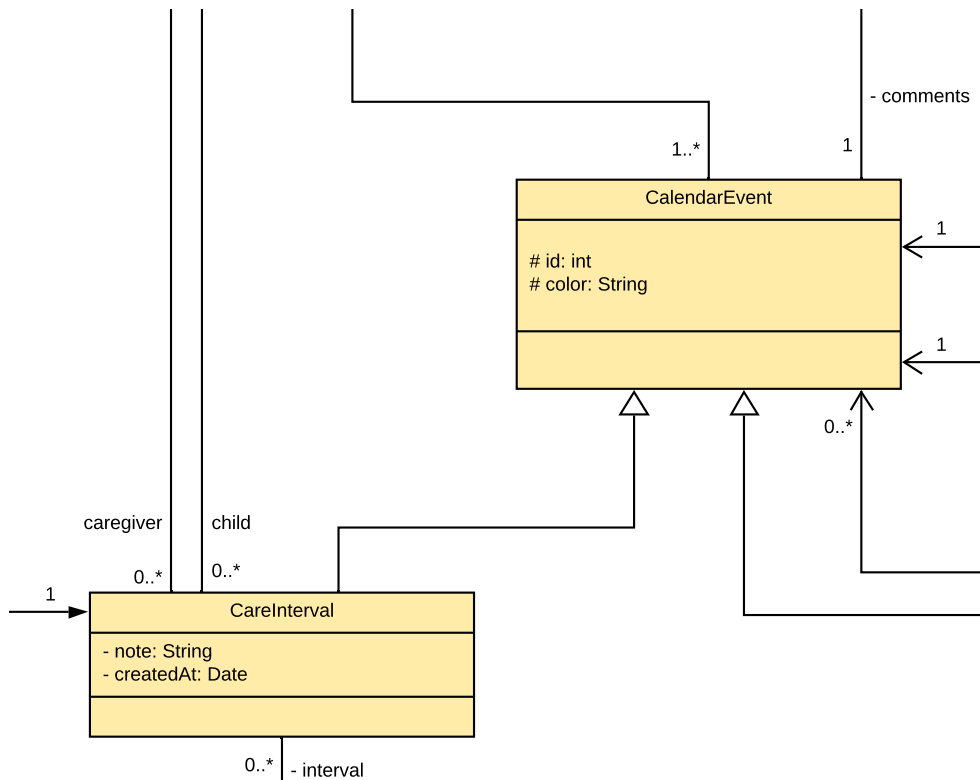


Obrázek 2.9: Návrh entit Alimony a AlimonySettings podle doménového modelu z předmětu BI-SP2

### 2.6.3 Pečovatelské dny

Než se začneme zabývat podrobným popisem problému, je potřeba popsat účel této entity a její využití backendem a frontendem. Tato entita reprezentuje jednorázový interval nebo opakující se interval pečovatelských dnů jednoho z rodičů nebo ostatních členů rodiny. Pro rozlišování pečovatelských dnů různých členů rodiny má každý uživatel přiřazenou vlastní barvu. Tato entita se používá pro dva účely. První a nejdůležitější účel je nastavení pečovatelských dnů pro rodiče. Nastavení musí pokrývat všechny dny kalendáře. Dítě by nemělo mít ve svém kalendáři den, který není označen žádnou barvou a zároveň by neměly vznikat konflikty. Druhým účelem jsou jednorázové změny pečovatelských dnů. Tyto změny jsou vyžadovány pro případy, kdy dlouhodobé nastavení kalendáře neplatí. Příkladem může být výlet dítěte k prarodičům. Během tohoto časového intervalu nejsou rodiče zodpovědní za dítě, proto je potřeba uvést jednorázovou změnu do kalendáře, která zaznamená tuto situaci. Jiným příkladem je předání péče jednoho rodiče druhému, což může nastat z různých důvodů.

Aktuální návrh pečovatelských dnů je navržen příliš komplikovaně (viz obrázek 2.10) a současně zahrnuje dva různé případy využití: jednorázové změny pečovatelských dnů členů rodiny a dlouhodobá nastavení pečovatelských dnů rodičů. Dlouhodobá nastavení pečovatelských dnů rodičů nevyžadují kompli-



Obrázek 2.10: Návrh pečovatelských dnů podle doménového modelu předmětu BI-SP2

kovaný návrh intervalů (viz sekci 2.6), proto je potřeba oddělit řešení tohoto problému. Také je potřeba přidat odkaz na konkrétního rodiče, který je zodpovědný za tento den pro zjednodušení práce s touto entitou v rámci frontendové části aplikace. Jednorázové změny pečovatelských dnů je druhý problém, který je potřeba oddělit. Návrh této entity je podobnější aktuálnímu návrhu, protože tyto změny mohou vyžadovat složitá pravidla opakování. Navržené změny a následné implementace budou popsány v sekci 3.1.3.

### 2.6.4 Oznámení

Dosavadní návrh oznámení reprezentuje upozornění uživatele o požadavku na změnu (viz obrázek G.2). Po provedení částečné implementace frontendové a backendové částí bylo zjištěno, že návrh oznámení potřebuje úprav ve dvou různých směrech. Prvním směrem je zavedení různých typu oznámení, například oznámení pomocí e-mailu nebo SMS zprávy. Současná implementace frontendové části aplikace vyžaduje zavedení typu upozornění, které by reprezentovaly upozornění v rámci Android aplikace. Tyto upozornění vyžadují

## 2. ANALÝZA

---

specifické nastavení entity, například možnost přečtení jednotlivé instance. Druhým směrem je rozšíření seznamu událostí, které mohou působit vzniku oznámení. Příčinami mohou být:

- změny pečovatelských dnů;
- alimenty;
- změny nastavení alimentů;
- potřeby dítěte;
- změny v rámci rodiny.

Podle požadavků frontendové části aplikace je také potřeba uvádět příčinu vzniku upozornění v každé instanci. Příčinou může být jedná z výše zmíněných událostí. Za účelem navržení kvalitnějšího API je také potřeba přidat možnost přecíst najednou všechna upozornění, která uživatel má. Kompletní řešení problému bude popsáno v sekci 3.1.4.

### 2.7 Analýza bezpečnosti

V této sekci budou popsány procesy, které by měly zaručovat bezpečnost aplikace ze strany serverového backendu.

#### 2.7.1 Role

Než se uživatel přihlásí do rodiny, nemá žádnou roli. Po přihlášení do rodiny má uživatel roli v rámci přihlášené rodiny (viz obrázek G.3), podle které se mohou lišit jeho přístupová práva. Hlavní rolí v aplikaci je rodič. Uživatel s takovou rolí má přístup ke všem potřebám dítěte a všem záznamům v kalendáři. Rodič také může vytvářet pozvání do rodiny pro libovolného uživatele a nastavit mu libovolnou roli, včetně role rodiče. Mimořádnou rolí v rámci systému je dítěte. Uživatel s takovou rolí nemůže vlastnit pečovatelský den nebo splnit přání. Přihlášení dítěte může proběhnout i bez vytvoření klasického uživatele v rámci systému. Ostatní uživatelé v rodině mají roli příbuzných.

#### 2.7.2 Autorizace

Návrh bezpečné aplikace nebyl cílem předmětů zmíněných v sekcích 2.1 a 2.2. Proto návrh a předešla implementace neobsahuje proces přihlašování uživatele do systému. Navržené procesy autentizace a autorizace budou podrobně popsány v sekci 3.3.

---

```
1 @RunWith(SpringRunner::class)
2 @SpringBootTest
3 class RozvodyApplicationTests {
4
5     @Test
6     fun contextLoads() {
7     }
8
9 }
```

---

Obrázek 2.11: Ukázka předešlého testování

## 2.8 Analýza testování

Dosavadní návrh neobsahuje informaci o implementaci testování. Ale předešlá implementace obsahuje pouze jeden test, který se zaměřuje na ověření, jestli se načte kontext aplikace<sup>16</sup> (viz obrázek 2.11). Cílem implementace testování aplikace bude provedení v rámci této bakalářské práce.

## 2.9 Průběžná integrace

Na začátku je potřeba popsat pravidla zavedená pro distribuovaný systém správy verzí – Git – před začátkem implementace serveru. Proces vývoje byl rozdělen do dvou hlavních větví. První větev reprezentuje aktuální verzi aplikace a je označena jako „master“, což je implicitní větev v rámci Git. Druhá větev je označena jako „dev“ a je určena pro proces vývoje. Pro implementaci jednotlivých úkolů je potřeba udělat kopii této větvi a po vyřešení úkolu je potřeba zařadit všechny změny zpět do větve `dev`. Takový přístup dává možnost pracovat na stejném projektu několika programátorům najednou a s jistotou uschovávat vlastní změny na server bez ohledu na jejich kompletnost.

Vedoucím této bakalářské práce a současně vedoucím tohoto projektu byl poskytnut server pro testování, dostupný z IP: „http://37.46.80.230/“. Větev `master` a `dev` se automaticky nasazují po aktualizaci. Aplikace uložená do větve `master` je dostupná na portu 8998. Aplikace uložená do větve `dev` je dostupná na portu 8778.

---

<sup>16</sup>Pokročilý kontejner, který funguje podobně jako `BeanFactory`. Načítá definice beanů, provazuje je a vydává v případě nutnosti.



---

## Návrh a implementace

V této kapitole budou popsány změny návrhu programu, které se skládají z navržených změn podle požadavků frontendového týmu a návrhu chybějících funkcí. Dále bude popsána implementace změn návrhu a chybějících funkcí. Nakonec bude popsán návrh a následná implementace funkcí, které jsou zaměřené na zlepšení kvality výsledného softwaru.

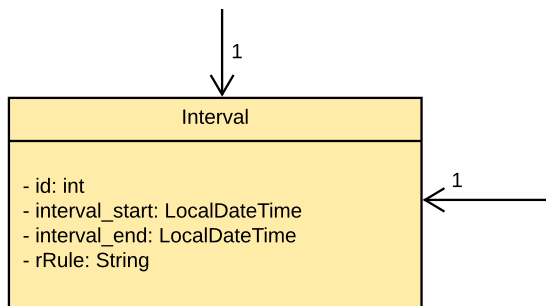
Při úpravách současného návrhu byl proveden velký počet změn různé významnosti. V této kapitole budou uvedeny klíčové změny. Při implementaci návrhu autorovi pomohlo přečtení doporučené literatury.[23]

### 3.1 Úpravy podle požadavků frontendového týmu

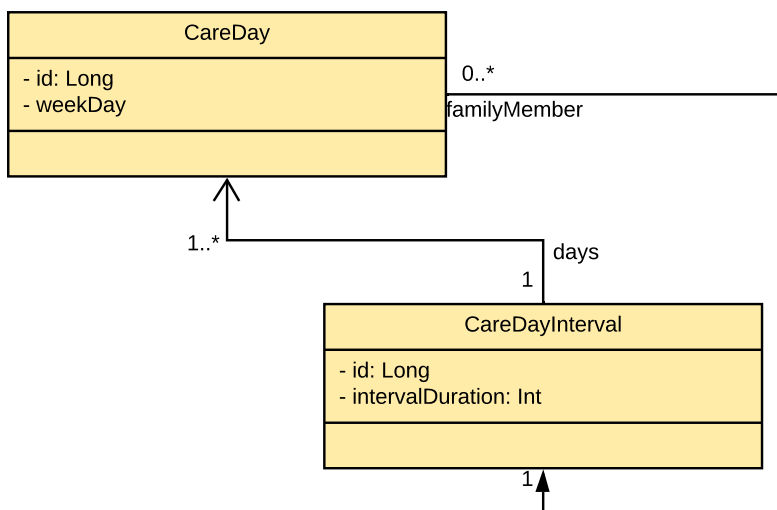
V této sekci budou popsány navržené změny podle požadavků frontendového týmu a jejich následná implementace.

#### 3.1.1 Interval

První změnou je nový návrh entity `Interval`. Původní návrh nevyhovoval svou složitostí a zároveň jenom částečným pokrytím možných případů užití (viz obrázek G.1). Podrobně tento problém byl popsán v sekci 2.6.1. Entita řešila několik samostatných problémů najednou: pravidla pečovatelských dnů rodičů a klasické intervaly definující pravidla opakování nebo jednorázovou událost. Proto entita `Interval` byla rozdělena do dvou samostatných částí. První část se zabývá pravidly pro pečovatelské dny rodičů (viz obrázek 3.1). Tento typ intervalu bude podrobně popsán v sekci 3.1.3. Ostatní případy použití intervalů nevyžadují, ani specifické atributy, ani specifický rozsah možných pravidel, proto ostatní případy použití řeší stejná entita – `Interval` – (viz obrázek 3.1).



Obrázek 3.1: Nový návrh entity Interval



Obrázek 3.2: Návrh entity CareDayInterval

Nový návrh entity **Interval** je kompilovaný, proto je potřeba podrobně popsat použitou architekturu. Nový návrh je postaven na jiném principu, kde časové rozmezí může být reprezentováno dvěma způsoby. Oba dva způsoby vyžadují uvedení začátku intervalu. Tento parametr je povinný. První způsob vyžaduje kromě začátku intervalu i konec intervalu. Takovým způsobem můžeme definovat jednorázový interval po sobě jdoucích dnů. Druhý způsob vyžaduje zadání pravidla opakování. Takto můžeme definovat stejný interval, ale mnohem složitějším způsobem. Na druhou stranu, pomocí takového pravidla můžeme definovat libovolně složitý interval. Například zmíněný v sekci 2.6.1 interval, který se bude opakovat každý poslední den měsíce. Výsledný návrh sestavení intervalu vyžaduje, aby byl zadán buď jenom konec intervalu, nebo aby bylo zadáno jenom pravidlo opakování. V případě, že tyto dva pa-



rametry budou zadány najednou, server vyhodí chybu a zastaví vytvoření ne-  
správného intervalu. Pravidlo opakování je reprezentováno pomocí textového  
řetězce, který má být zadán podle standardu RFC 5545.[24] Pro pohodlné  
sestavení pravidel opakování při testování aplikace byl navržen a implemen-  
tován interní DSL jazyk<sup>17</sup>. Tento jazyk bude podrobně popsán v sekci 3.2.1.

---

```
1 @Scheduled(cron = "\${scheduled.alimonyFactory.cronExpression}")
2 fun createAlimonyForEachAlimonySetting() {
3     logger.info("Started scheduled alimony creation process;")
4     val alimony = alimonySettingRepository
5         .findAllByEnabledTrue()
6         .map { it.createAlimony() }
7     if (alimony.isEmpty()) {
8         alimonyRepository.saveAll(alimony)
9     } else {
10        logger.debug(
11            "AlimonyFactory didn't find any AlimonySetting to work with;"
12        )
13    }
14 }
```

---

Obrázek 3.3: Ukázka metody vytvářející instance alimentů

#### 3.1.2 Alimenty

Druhou důležitou úpravou současného návrhu je změna entity `Alimony`. Tyto úpravy se dotýkají entity samotné a spravování instancí této entity.

##### 3.1.2.1 Úprava entity

Za účelem zvýšení samostatnosti instancí entity `Alimony` byla přidána závislost na entitu `Family`. Také byl přidán atribut `value`, který kopíruje z entity `AlimonySetting` částku, která má být uhrazena jedním z rodičů. Příčinou přidání tohoto atributu je existence možnosti aktualizace nastavení alimentů, což působí zneplatnění již existujících instancí alimentů. V případě, že rodiče změni hodnotu v nastavení alimentů, ztratí se možnost zjistit hodnoty alimentů, které byly vytvořeny do změny.

---

<sup>17</sup>DSL jazyk využívající obecný programovací jazyk.

### 3. NÁVRH A IMPLEMENTACE

---

```
1 \# Cron expression: at 01:01 AM on the 1st day of every month
2 scheduled.alimonyFactory.cronExpression=0 1 1 1 * ?
```

---

Obrázek 3.4: Ukázka konfigurace cron výrazu pro plánování vytváření alimentů

```
1 scheduled.alimonyFactory=true
```

---

Obrázek 3.5: Ukázka proměnné prostředí zapínající `AlimonyFactory`

#### 3.1.2.2 Spravování alimentů

V této sekci bude popsán návrh třídy, která se věnuje pravidelnému vytváření alimentů.

Všechny instance alimentů je potřeba vytvářet pravidelně. Proto vytváření bylo objednáno do jednoho procesu, který je stejný pro všechny instance. Proces vytváření alimentů zajišťuje specifická metoda, která se nachází ve třídě `AlimonyFactory` (viz obrázek 3.3). Metoda vyhledává všechny aktivní instance `AlimonySetting` a pro každou vytvoří instanci entity `Alimony`.

Vytváření alimentů má být pravidelné a nemusí vyžadovat účast člověka. Proto byla využita funkcionality frameworku Spring, která poskytuje možnost plánování automatického spouštění.[25] Anotace `@Scheduled` vytváří z dané metody komponentu, která se vykonává podle uvedeného pravidla. Existuje několik podporovaných typů pravidel. Například nastavení intervalu mezi nastartováními nebo mezi ukončeními. Ale pro plánování vytváření alimentů je potřeba nastavit konkrétní den měsíce nebo roku, který se bude opakovat. Proto bylo zvoleno cron pravidlo, které je také touto podporováno anotací.[26] Tento výraz je textovou řádkou, která se skládá z čísel oddělených mezerami reprezentujícími čas spouštění. Toto pravidlo nebylo zadáno přímo do kódu. Pro přidání možnosti pohodlné konfigurace pravidla byla definována proměnná prostředí, která se nachází v konfiguračním souboru příslušného profilu (viz obrázek 3.4). Implementace byla provedena podle návodu z knihy [27]. Podrobněji budou profily popsány v sekci 3.4.

Byla přidána možnost měnit čas spouštění metody pomocí konfiguračního souboru, ale také bychom potřebovali mít možnost vypnout tuto metodu, například pro robustní testování. Proto byla přidána třída, která vytváří instanci tříd obsahující automaticky spustitelné metody. Třída je definována jako komponenta frameworku Spring a každá její metoda je určena pro vy-

tváření instancí jiných tříd podle předem definované podmínky. V případě, že podmínka má hodnotu `false`, příslušná třída se nevytváří. Pro vytváření třídy `AlimonyFactory` byla definována podmínka, která ověřuje, jestli proměnná `scheduled.alimonyFactory` je nastavená na hodnotu `true` (viz obrázek 3.5). V případě, že tato proměnná nebyla definována pro aktuální profil aplikace, proměnná se automaticky nastaví na hodnotu `false` a instance třídy `AlimonyFactory` nebude vytvořena.

#### 3.1.3 Pečovatelské dny

V předchozím návrhu implementace dlouhodobých nastavení pečovatelských dnů pro rodiče a jednorázových změn byly reprezentované pomocí stejné entity (viz obrázek 2.10). Problém je právě ve sjednocení několika problémů do jednoho. Proto bylo rozhodnuto rozdělit řešení do dvou částí. Výsledný návrh se skládá ze čtyř entit (viz obrázek G.4):

- `CareDaysSetting`, reprezentující dlouhodobé nastavení pečovatelských dnů;
- `CareDayInterval`, reprezentující interval pečovatelských dnů pro dlouhodobé nastavení;
- `CareDay`, reprezentující jeden pečovatelský den pro dlouhodobé nastavení;
- `CareDayChange`, reprezentující změnu pečovatelských dnů.

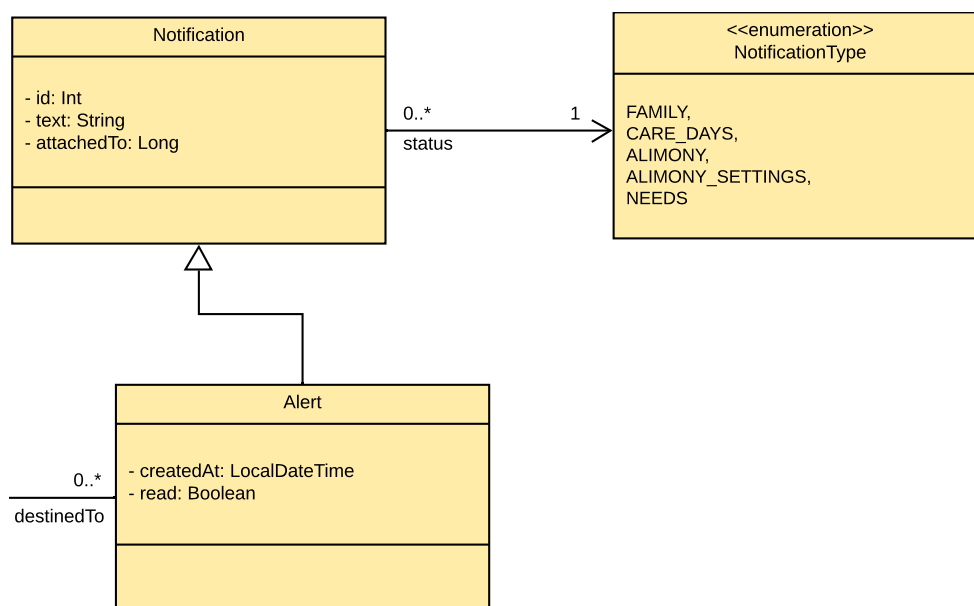
Dlouhodobá nastavení pečovatelských dnů rodičů již byla zmíněna v sekci 3.1.1. Nastavení se skládají z po sobě jdoucích dnů, kde každý den má odkaz na jednoho rodiče, který má v péči dítě v tento den. Interval je uchovávan v entitě `CareDaysSetting`, která reprezentuje nastavení pečovatelských dnů rodičů pro konkrétní rodinu. Tato nastavení definují implicitní nastavení pečovatelských dnů, podle kterých se vyplní kalendář rodiny.

Změny pečovatelských dnů jsou reprezentovány entitou `CareDayChange`. Interval změny nebo pravidlo opakování změny je reprezentován pomocí entity `Interval`. Každá změna se vztahuje ke konkrétnímu členovi rodiny a instanci nastavení pečovatelských dnů příslušné rodiny.

#### 3.1.4 Oznámení

Předchozí návrh oznámení nebyl vhodný pro frontendovou část aplikace. Oznámení byla určena jenom pro konkrétní případ užití – oznámení o požadavku na změnu. Byla navržena obecně a neměla konkrétní typ. Tudíž oznámení elektronické pošty a upozornění v telefonu byla reprezentována pomocí stejné entity.

### 3. NÁVRH A IMPLEMENTACE



Obrázek 3.6: Nový návrh entity `Notification` a na ně se navazujících

Předchozí abstraktní entita byla určena pro definování základních údajů. Zděděná entita definovala důležitost tohoto oznámení pro konkrétního uživatele a uživatele samotného. Bylo rozhodnuto zachránit abstraktní návrh entity `Notification`, ale změnit jeho účel. Abstraktní entita bude obsahovat základní údaje pro všechny typy oznámení. Zděděné entity budou reprezentovat konkrétní typ oznámení. Současná implementace frontendové části aplikace vyžaduje zavedení typu upozornění, které by reprezentovaly upozornění v rámci Android aplikace. Proto byla přidána entita `Alert` (viz obrázek 3.6), která dědí základní údaje od entity `Notification`. Pro zaručení konkrétní příčiny vytvoření oznámení byl přidán odkaz na událost, která je důvodem vzniku tohoto oznámení, do abstraktní entity a také typ této události. Typ oznámení je reprezentován pomocí entity výčtového typu – `AlertCause`. Každá instance entity `Alert` obsahuje atribut `read` označující, jestli je oznámení přečteno uživatelem. Pro zaručení kvalitního návrhu API byla přidána možnost přečtení všech upozornění najednou. Přečtení upozornění je reprezentováno pomocí přidání atributu `read` hodnoty `true`.

Výsledný návrh oznámení reprezentuje jednotlivá oznámení, která jsou dostupná pouze pro konkrétního uživatele. Přidané komentáře do oznámení jsou viditelné pouze pro adresáta příslušného oznámení, proto proces přidávání komentářů byl odstraněn. Také byl přidán atribut `text`, který je nutným pro uvádění podrobnějšího popisu oznámení.

---

```
1  /**
2  * Valid Interval with recurrence rule.
3  */
4  private val validInterval = Interval(
5      id = 1,
6      interval_start = creationTime,
7      interval_end = null,
8      rRule = rule(frequency = Frequency.WEEKLY, count = 10) {
9          byDays {
10             and(DayOfWeek.MONDAY)
11             and(DayOfWeek.WEDNESDAY)
12             and(DayOfWeek.SUNDAY)
13         }
14     }
15 )
```

---

Obrázek 3.7: Ukázka instance entity `Interval` s pravidlem opakování

## 3.2 Navržené změny

V této sekci budou popsány změny navržené autorem této práce zaměřené na vylepšení výsledné aplikace.

### 3.2.1 Interní DSL jazyk

Pro pohodlné testování entit závislých na entitě `Interval` byl implementován interní DSL<sup>18</sup> jazyk, poskytující možnost vytvořit pravidlo opakování. Na obrázku 3.7 je zobrazen příklad intervalu, který má pravidlo opakování vytvořené pomocí DSL jazyka. Příklad je převzat ze současné implementace testování.

### 3.2.2 Album

Libovolný člen rodiny může mít několik vlastních alb, která potřebuje rozlišovat mezi sebou. Předchozí návrh entity `Album` neposkytoval možnost přidání názvu, proto do entity byl přidán příslušný atribut.

### 3.2.3 Konverter

V sekci 2.5 již bylo zmíněno, že pro implementaci serveru byla zvolena třívrstvá architektura. Doménová vrstva komunikuje s datovou vrstvou pomocí

---

<sup>18</sup>Domain Specific Language

### 3. NÁVRH A IMPLEMENTACE

---

```
1 internal interface InterfaceConverter<E : InterfaceEntity, D : InterfaceDTO> {  
2  
3     fun convertToEntity(dto: D): E  
4  
5     fun convertToDTO(entity: E): D  
6  
7 }
```

---

Obrázek 3.8: Ukázka rozhraní `InterfaceConverter`

tříd reprezentujících entity databáze, ale s aplikační vrstvou tato komunikace probíhá pomocí DTO. Konverze mezi entitou a DTO se provádí v této vrstvě. Za konverzi z entity na DTO odpovídá třída entity. Za konverzi z DTO na entitu odpovídá třída DTO. Takový postup těsně provazuje entitu s příslušnou třídou DTO, tento problém se také nazývá *tight coupling*. Zavedení konvertorů zmenšuje takovou závislost. Každý konvertor obsahuje metodu konvertující entitu na příslušné DTO a metodu konvertující DTO na příslušnou entitu. Takový postup umožňuje odstranit stejné metody z příslušných tříd, což zvětšuje modularitu návrhu. Pak je možné snadno přidat jiné DTO reprezentující stejnou entitu bez úprav entity samotné, a naopak.

Pro dosažení kvalitnějšího výsledku a zrychlení implementace bylo zavedeno rozhraní definující implicitní metody pro každý konvertor (viz obrázek 3.8). Rozhraní také potřebuje zadání typu entity a DTO, které mají být podtřídami `InterfaceEntity` a `InterfaceDTO`. Tyto rozhraní nedefinují základní funkce, ale jenom označují typy těchto tříd.

#### 3.2.4 Našeptávač pro výjimky

Našeptávač pro výjimky byl zmíněn již v sekci 2.5.1. Tato komponenta byla rozšířena o typ výjimky označující, že DTO nebyl správně serializován kvůli nedostatku potřebných atributů. Příkladem může být překlep v názvu povinného atributu. Také návratová zpráva byla rozšířena o textový název chyby a časové razítko. Výslednou konfiguraci našeptávače najdete v příloze F.

#### 3.2.5 Aplikační vrstva

V předchozí implementaci byl použit POST požadavek, jak pro vytváření, tak pro aktualizaci. Tento požadavek očekává Data Transfer Object (DTO) v těle požadavku. Požadavky se rozlišují mezi sebou přítomností identifikátoru entity v DTO. Pokud je ID nastaveno na `null` vytvoří se nový záznam v databázi. V opačném případě se server pokusí najít a aktualizovat záznam.

Nový návrh aplikační vrstvy podporuje POST a PUT požadavky. POST požadavek je určen pro vytvoření nového záznamu a vyžaduje aby atribut `id` v DTO byl nastaven na hodnotu `null`. PUT požadavek je určen pro aktualizaci záznamu a vyžaduje aby atribut `id` v DTO byl nastaven na číslo.

Jeden uživatel může být současně členem několika různých rodin. Serverový backend nemůže předem vědět, ve které rodině se aktuálně uživatel nachází. Proto do API byly přidány požadavky, které vyžadují zadání identifikátoru konkrétní rodiny. Na základě tohoto identifikátoru se filtrují záznamy, které obdrží uživatel. Požadavky nevyžadující identifikátor rodiny nebyly odstraněny, protože všechny požadavky procházejí proces filtrace. Následně konečný uživatel obdrží jenom záznamy, které jsou pro něj dostupné.

Dokumentace aplikační vrstvy byla též rozšířena o návratové statusy pro každou metodu. Všechny parametry metod jsou získány z požadavků, proto byly přidány i podrobné popisy parametrů metod.

### 3.2.6 Doménová vrstva

Nový návrh aplikační vrstvy rozděluje proces vytváření nových záznamů a proces aktualizace již existujících záznamů. Proto tento proces byl rozdělen do dvou separátních procesů i v doménové vrstvě.

Proces filtrování záznamů podle aktuálně přihlášeného uživatele se provádí na datové vrstvě. Doménová vrstva jenom používá specifické metody datové vrstvy obsahující filtraci. V případě, že požadovaná data nepatří konkrétnímu uživateli, ale celé rodině, provádí se ověření, jestli aktuálně přihlášený uživatel patří do této rodiny.

### 3.2.7 Datová vrstva

V předchozí implementaci datové vrstvy byly použity implicitní metody, které poskytuje framework Spring Data. V současné implementaci byly přidány vlastní metody s použitím nástroje pro vytváření metod dotazování (*query methods*). Přidané metody jsou zavedeny za účelem filtrování záznamů, které obdrží uživatel. Filtrování se provádí na základě údajů o aktuálně přihlášeném uživateli. V případě, že požadované záznamy nepatří konkrétnímu uživateli, ale celé rodině, ověřuje se, jestli aktuálně přihlášený uživatel patří do této rodiny.

### 3.2.8 Implementace historie změn

Důležitou částí aplikace je zaznamenání změn udělaných uživatelem. Úpravou je změna záznamu entity, kterou uživatel s rolí „ROLE\_USER“ může upravit.

### 3. NÁVRH A IMPLEMENTACE

---

Takovou roli má jakýkoliv uživatel přihlášený do systému. Aktivita `root`<sup>19</sup> uživatele není uvažována, protože tento uživatel není přítomný v rámci profilu pro produkci. Příkladem takové změny může být změna nastavení alimentů nebo označení oznámení jako přečtené.

Historie změn ještě nebyly implementovány, ale návrh již existoval a hlavní myšlenka zůstává stejná. Záznam historie kopíruje všechny atributy entity a přidává čas vytvoření záznamu a vlastní identifikátor<sup>20</sup>. Entita `History` je abstraktní třídou, neboli generalizací v případě návrhu v doménovém modelu. Jednotlivé entity, které jsou zděděné od této entity, reprezentují historie příslušných entit (viz obrázek G.6).

Tato entita byla implementována jako poslední a všechny změny, které se jí týkají, jsou srozumitelné jenom po přečtení všech změn návrhu. Nejprve bylo potřeba upravit návrh podle implementovaných změn v příslušných entitách a až poté začít s návrhem vhodných změn této entity (viz obrázek G.7).

Dalším krokem úprav je navržení úprav, které už nejsou nutné, ale výrazně zlepšují výsledný návrh programu:

- přidání do každé entity reprezentující historii odkazy na příslušné entity, například entita `AlimonyHistory` bude mít navíc odkaz na entitu `Alimony`;
- zavedení nové entity historie `CareDaysSettingHistory` podle návrhu entity `CareDaysSetting`, která byla popsána v sekci 3.1.3.

Výsledný návrh entity `History` (viz přílohu E) byl kompletně implementován. Také byly přidány řadiče pro každý typ historie, které poskytují možnost vyhledat záznamy. Uživatel nemůže vytvořit nový záznam historie nebo změnit již existující záznam. Proto API serveru nepodporuje příslušné požadavky. Vytvoření instancí historie se provádí serverem automaticky při aktualizacích příslušných entit. Spravování entit historie v doménové vrstvě bylo rozděleno do dvou logických bloků. První blok se zabývá spravováním požadavků Android aplikace. Každý typ historie má vlastní rozhraní reprezentující metody, které je možné provádět s touto entitou. Druhým logickým blokem je proces vytváření instancí historie. Za tímto účelem bylo přidáno rozhraní `InternalHistoryService`, které poskytuje metodu `toHistory` pro uložení instance do databáze. Metoda očekává jako parametr instanci třídy před provedením změn a konvertuje tuto instanci do příslušné instance historie. Každý typ historie má též vlastní konvertor z entity do entity historie. Například,

---

<sup>19</sup>Uživatel s rolí „ROLE\_ROOT“.

<sup>20</sup>Identifikátor v rámci databáze.



po vložení do metody `toHistory` parametru typu `Bill`, bude zavolán konvertor `BillToHistoryConvertor`, který vrátí instanci třídy `BillHistory`.

## 3.3 Návrh bezpečnosti

### 3.3.1 OAuth 2.0

Za účelem zabezpečení procesu autorizace byl zvolen protokol OAuth 2.0. Tento protokol je nativně podporován frameworkem Spring. Proto byla provedena jeho konfigurace, která vyžadovala implementaci následujících rozhraní:

- `UserDetailsService` – pro definování zdroje informací o uživateli;
- `WebSecurityConfigurerAdapter` – pro definování implementace rozhraní `UserDetailsService` a nastavení kontextu bezpečnosti;
- `AuthorizationServerConfigurerAdapter` – pro nastavení procesu přihlašování;
- `ResourceServerConfigurerAdapter` – pro nastavení přístupu do řadičů.

Rozhraní `UserDetails` bylo implementováno pro každý profil zvlášť. Pro pohodlný proces vývoje byl přidán uživatel s rolí „ROLE\_ROOT“ pro profil `dev`. Za účelem testování byly přidány implicitní uživatele pro profil `test`. Třetí implementace byla provedena pro profil produkce, proto žádného implicitního uživatele tato implementace nemá.

### 3.3.2 HTTPS

V předchozí implementaci aplikace byl použit protokol HTTP<sup>21</sup>, který nezaručuje bezpečnou komunikaci mezi klientem a serverem. Proto tento protokol byl nahrazen protokolem HTTPS<sup>22</sup>, který vyžaduje certifikát podepsaný třetí stranou. Tento certifikát je uložen v aplikaci.

## 3.4 Profily

Během implementace serveru byla potřeba rozšířit návrh profilů aplikace. Původní návrh obsahoval jenom implicitní profil a profil vývoje, obsahující konfiguraci databáze H2. Výsledný návrh aplikace potřebuje více profilů, proto byly přidány dva další profily: profil pro produkci a profil pro testování. Důvodem přidání dalších profilů jsou rozdíly mezi konfiguracemi, které potřebují

---

<sup>21</sup>Hypertext Transfer Protocol

<sup>22</sup>Hypertext Transfer Protocol Secure

konkrétní případy spouštění aplikace. Princip chování profilu byl již zmíněn v sekci 2.5.3, proto v této sekci budou popsány provedené změny.

Pro podrobnější popis přidávaných profilů je potřeba nejdříve popsat výslednou konfiguraci existujícího profilu vytvořeného pro proces vývoje. Seznam provedených změn a rozšíření, které navazují na tento profil:

- Rozšířena konfigurace databáze. Změny byly provedeny za účelem přesnější definice konfigurace (viz sekci 3.5);
- Přidána nutná konfigurace pro bezpečnost aplikace, která byla popsána v sekci 3.3;
- Přidána proměnná do konfiguračního souboru, která zapíná pravidelné vytváření alimentů (viz sekci 3.1.2) a také proměnná definující pravidlo, podle kterého se řídí plánování spouštění této třídy;
- Přidána specifická implementace pro rozhraní *UserDetails* (viz sekci 3.3);
- Přejmenován profil „development“ na „dev“, za účelem lepší přehlednosti kódu při definování několika profilů komponenty najednou.

Výše uvedená konfigurace je vhodná pro vývoj aplikace, ale zároveň není vhodná pro produkci a testování aplikace. Proto bylo rozhodnuto nechat v implicitním konfiguračním souboru jenom definici profilu, který by měla aplikace využít pro následující spuštění, a přidat další profily podle případů užití.

Prvním profilem, který byl přidán, je profil pro produkci. Hlavním rozdílem tohoto profilu od profilu vývoje je použitá databáze. Podrobněji bude zvolená databáze popsána v sekci 3.5. Pro tuto sekci je postačující zmínit, že zvolená databáze je PostgreSQL. Druhou odlišností je jiná implementace rozhraní *UserDetails*. Produkční verze aplikace by neměla mít `root` uživatele. Jiný konfigurační soubor také dovoluje nechat konfiguraci pro produkci nedotčenou při možném velkém počtu změn v jiných profilech. Například, můžeme nechat vypnuté plánované vytváření alimentů pro vývojový profil.

Druhým profilem, který byl přidán do aplikace, je profil pro testování. Změny se dotkly stejných aspektů jako profilu produkce. Byla přidána nová implementace rozhraní *UserDetails*, která obsahuje implicitních uživatele s různými rolami. Také byla přidána konfigurace databáze. Pro tento profil byla zvolena stejná databáze jako i pro profil produkce. Takový postup zmenšuje šance na výskyt nečekaných chyb, působených použitím odlišné databáze v produkci.

## 3.5 Databáze

Použité databáze byly popsány v sekci 1.7. V této sekci bude vysvětleno, proč byly zvoleny tyto databáze.

Profil pro vývoj používá databázi H2. Stejná databáze byla použita při předchozí implementaci. Databáze nevyžadovala její nahrazení jinou databází, protože nevyžaduje nastartování databáze zvlášť od aplikace, ale nastartuje se spolu s aplikací. Všechna data se ukládají přímo v paměti aplikace, proto po restartování aplikace smaže všechny vytvořené záznamy.

Pro profil produkce bylo rozhodnuto použít jinou databázi. Volilo se mezi dvěma databázemi: PostgreSQL a MySQL. Tyto databáze jsou jedny z nejpoužívanějších databází v současné době a zároveň jsou otevřeným softwarem. Po porovnání těchto dvou databází, byla vybrána databáze PostgreSQL. Některé funkce byly považovány za důležité přesto, že ještě nebyly využity v implementaci a návrhu. Příklady převah PostgreSQL nad MySQL:

- PostgreSQL je objektově relační databáze [28], zato MySQL je pouze relační databáze. To znamená, že PostgreSQL dovoluje dědění tabulek a přetěžování metod;
- PostgreSQL poskytuje možnost definovat vlastní typy [29];
- autor této práce má zkušenosti s PostgreSQL.

Informace o rozdílech těchto databází byly převzaty z několika článků na internetu. Nejpoužitelnějšími pro autora této práce byly [30, 31].

## 3.6 Návrh testování

Pro testování kódu budou využity frameworky JUnit 5 a Spring. Tyto frameworky byly podrobně popsány v sekci 1.5. Testování se skládá s `unit`<sup>23</sup> testů a `integračních`<sup>24</sup> testů. Návrhy testů se budou podstatně lišit mezi sebou. Proto budou využité funkce poskytované frameworkem JUnit 5, který od verze 5 poskytuje možnost označovat testy pomocí tagů.[32] Podrobnější informace o implementaci tagů budou uvedeny v sekci 4.1. Kompletní implementace testování bude popsána v kapitole 4.

---

<sup>23</sup>Testy, zaměřené na ověření správnosti fungování samostatně testovatelných částí programu.

<sup>24</sup>Testy, zaměřené na ověření správné komunikace mezi komponentami.

### 3.6.1 Unit testy

Unit testy využívají pouze funkce frameworku JUnit 5 a testují základní funkce. Otestovány mají být třídy a metody, které mohou být otestovány samostatně. Tyto testy také mají být označeny jako unit testy pomocí tagů.

### 3.6.2 Integrované testy

Integrované testy budou testovat jednotlivé řadiče. Tyto testy jsou navrženy pomocí principu IoC a budou označeny jako integrované testy a zároveň jako pomalé testy.

## 3.7 Implementace bezpečnosti

V současné implementaci je autorizační server součástí serverového backendu a odpovídá jak za registraci uživatele, tak i za přihlášení. Pro registraci má uživatel poslat DTO se svými údaji na příslušný řadič<sup>25</sup>. Pro přístup do zmíněného řadiče se používá typ přístupu *user credentials*. Uživatel potřebuje uvést adresu pro obdržení *tokenu*<sup>26</sup>, identifikátor uživatele a *secret*. Potom uživatel obdrží *tokeny* umožňující provést registraci. Pro přihlašování uživatele se používá typ obdržení průkazů *resource owner password credetials*, kde uživatel potřebuje, navíc od předchozího typu, poslat svoje uživatelské jméno a heslo. Potom uživatel obdrží *access token*<sup>27</sup> a *refresh token*<sup>28</sup> jako návratové hodnoty. Platnost *tokenu* pro komunikaci je omezen jednou hodinou. Platnost *tokenu* pro obnovení *access tokenu* pro komunikaci je omezen jedním týdnem.

---

<sup>25</sup>Řadič je namapován na cestu „/api/v1/aut“.

<sup>26</sup>Alfanumerické heslo.

<sup>27</sup>Tento *token* klient používá pro komunikaci se serverem.

<sup>28</sup>Tento *token* klient používá pro obnovení *access tokenu* po vypršení jeho platnosti.

---

# Testování

V této kapitole bude popsán proces testování a doplňující funkce zjednodušující proces testování. Také bude popsáno pokrytí kódů testy.

## 4.1 Tagy

Tagy už byly zmíněny v sekci 3.6, ale jejich chování nebylo podrobně popsáno. Nyní si tedy popíšeme chování tagů a proč je potřebujeme.

Tagy označují jednotlivé testy nebo celé testovací třídy pomocí obyčejného textového řetězce (viz obrázek 4.1). Obsah řetězce může být libovolný. Za účelem pohodlnějšího využití tagů byly zvoleny řetězce, které popisují konkrétní aspekt chování testu, například pomalý test nebo integrační test. Při spouštění testů pomocí frameworku JUnit můžeme definovat testy, které bychom potřebovali spustit pomocí těchto předem definovaných tagů.

Stejného výsledku bychom mohli dosáhnout pomocí třídění testů do různých složek, ale podstatný rozdíl mezi těmito postupy je v tom, že jeden test může mít několik tagů najednou a přidání nebo odstranění tagu je jednoduché. Například test může být anotován jako integrační test a zároveň jako pomalý test proto, že provádí databázovou transakci.

---

```
1 @Tags(  
2     Tag("integration_test")  
3 )
```

---

Obrázek 4.1: Příklad tagu frameworku JUnit 5

## 4. TESTOVÁNÍ

---

```
1  /**
2   * Specifies test as a Integration test.
3   */
4  @Target(allowedTargets = [
5      AnnotationTarget.FUNCTION,
6      AnnotationTarget.ANNOTATION_CLASS,
7      AnnotationTarget.CLASS
8  ])
9  @Retention(AnnotationRetention.RUNTIME)
10 @MustBeDocumented
11 @Tags(
12     Tag("integration_test")
13 )
14 annotation class IntegrationTest
```

---

Obrázek 4.2: Příklad třídy `Annotation` zaměřující tag s textem „integration\_test“

Tagy jsou jenom textové řetězce, je zde tedy velká pravděpodobnost překlepu při implementaci velkého počtu testů<sup>29</sup>. Proto byly definovány vlastní anotace s předem definovanými tagy:

- `IntegrationTest`, který má tag obsahující text „integration\_test“;
- `SecurityTest`, který má tag obsahující text „security\_test“;
- `SlowTest`, který má tag obsahující text „slow\_test“;
- `UnitTest`, který má tag obsahující text „unit\_test“.

Anotace mají stejnou implementaci a liší se pouze pomocí textu uvnitř tagů. Kotlin poskytuje podporu pro vytváření vlastních anotací, proto byla využita vestavěná třída `annotation` (viz obrázek 4.2). Výsledné anotace jsou implementovány takovým způsobem, že pomocí nich je možné anotovat nejen testové metody, ale i celé třídy. Pak se anotace aplikuje na každý test, který obsahuje anotovaná třída. Také je možné anotovat i jiné anotace.

Pro spouštění testů je potřeba využít nástroj pro automatizaci sestavování programu – Gradle, který byl podrobně popsán v sekci 1.3 . Testy jsou rozděleny podle jejich typu pomocí tagů, proto byla přidána možnost, nejenom spustit všechny testy najednou, ale i spustit jenom unit testy nebo jenom integrační

---

<sup>29</sup>V době odevzdání bakalářské práce bylo implementováno 126 testů v 25 třídách.

---

```
1 tasks.test {
2     useJUnitPlatform()
3     testLogging {
4         events 'PASSED', 'FAILED', 'SKIPPED'
5     }
6
7     maxHeapSize = "1G"
8 }
9
10 task unitTests(type: Test) {
11     useJUnitPlatform {
12         includeTags 'unit_test'
13     }
14     testLogging {
15         events 'PASSED', 'FAILED', 'SKIPPED'
16     }
17 }
18
19 task integrationTests(type: Test) {
20     useJUnitPlatform {
21         includeTags 'integration_test'
22     }
23     testLogging {
24         events 'PASSED', 'FAILED', 'SKIPPED'
25     }
26 }
```

---

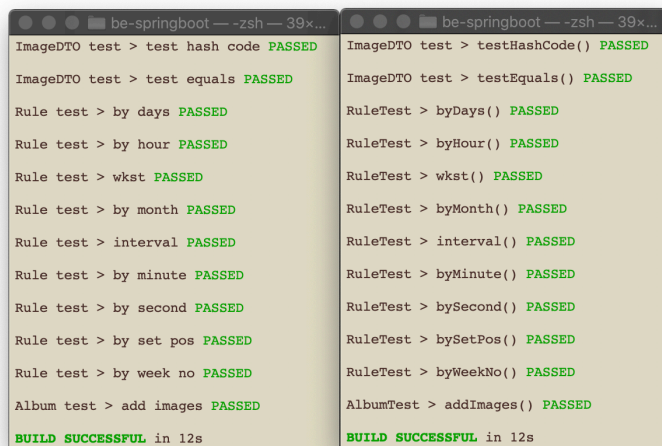
Obrázek 4.3: Úkoly pro spuštění testů

testy (viz obrázek 4.3). Kombinací tagů je mnohem více, ale konfigurační soubor obsahuje jenom konfigurace, které potřeboval autor této práce během vývoje. Přidat další konfigurace však není složité. Je potřeba pouze okopírovat předchozí konfiguraci v konfiguračním souboru a změnit seznam tagů podle potřeby. Seznam tagů je také možné doplnit. Všechny aktuální tagy se nacházejí ve složce `src/test/kotlin/cz/cvut/fit/sp/rozvody/annotation/tag`.

Spuštění testů není omezené příkazovým řádkem. Testy je také možné spustit pomocí vývojového prostředí nebo jiných nástrojů. Je možné použít například vývojové prostředí IntelliJ IDEA.

## 4. TESTOVÁNÍ

---



Obrázek 4.4: Příklad vypisování testů pomocí generátoru (4.4a) a bez využití generátoru (4.4b)

### 4.2 Zobrazování testů

Správné a pochopitelné zobrazování názvů testů zjednodušuje proces testování. Proto před začátkem implementaci testů bylo navrženo pravidlo pro zobrazování testů. Název testu má obsahovat stručný popis toho, co by měl tento test otestovat.

Pro čitelnější zobrazování byl implementován generátor transformující jméno testu nebo jméno třídy do čitelnější podoby (viz obrázek 4.4). Princip fungování je založen na překladu názvu z *camel case* na obyčejný text s mezerami. Výsledné názvy testů nebo tříd neobsahují žádné zbytečné symboly<sup>30</sup>. Pro pohodlné využití generátoru byla implementována anotace, která je aplikovatelná na třídy a metody. Anotace aktivuje generátor pro všechny testy, které třída obsahuje nebo pro konkrétní test, na který byla aplikována.

V případě, že název metody obsahuje text, který by se neměl korektně transformovat, potřebujeme použít anotaci `@DisplayName` a v závorkách této anotace uvést text, který by měl být zobrazen. Tento název přepisuje výsledek generátoru, proto může být tato anotace použita ve třídě, která má anotaci

<sup>30</sup>Za zbytečné symboly jsou považovány prázdné závorky na konci názvu metody atd.



generátoru. Příkladem takového nevhodného názvu může být třída se jménem `ImageDT0`, která bude přeložena na „Image d t o“, což není požadovaným výsledkem.

### 4.3 Unit testy

Unit testy jsou zaměřené na otestování samostatně testovatelných metod a tříd. Všechny unit testy jsou anotovány jako unit testy (viz sekci 4.1) za účelem přidání možnosti spustit tyto testy zvlášť od ostatních testů. Žádný z testů není závislý na celém kontextu aplikace nebo její části, proto jsou všechny unit testy rychlé.

Aplikace má malý počet samostatně testovatelných funkcí, proto bylo rozhodnuto se především zaměřit na kvalitní testování pomocí integračních testů, které budou popsány v následující sekci. Současná testování obsahuje 20 unit testů. Popis těchto testů je k nalezení v příloze D.1.

### 4.4 Integrační testy

Velká pozornost byla věnovaná integračním testům. Tyto testy ověřují, jestli jednotlivé komponenty aplikace pracují podle jejich návrhu. Testy jsou anotovány jako integrační testy (viz sekci 4.1) za účelem přidání možnosti spustit tyto testy zvlášť od ostatních, stejně jako unit testy.

Integrační testy vyžadují nahrání celého kontextu aplikace nebo jeho částí, proto jsou pomalejší než unit testy. Některé testy navíc mění kontext aplikace a vyžadují zrušení kontextu po dokončení jejich běhu. Takové testy jsou anotovány jako pomalé testy<sup>31</sup>.

Většina integračních testů je zaměřena na ověření správného fungování vrstvy řadičů (*controller*). Testování se provádí pomocí nástroje `MockMvc`, který poskytuje framework `Spring`.<sup>[33]</sup> Nástroj umožňuje otestovat funkcionalitu aplikace bez jejího úplného nastartování. Příklad využití nástroje je na obrázku 4.5. Tento kód má za úkol otestovat, zda řadič správně vytváří instanci entity *Comment*. Kontrola správnosti atributu se provádí na základě serializované verze DTO, který se vrátí jako návratová hodnota.

V rámci integračních testů byla také ověřena přístupová práva (viz sekci 3.3). Při testování řadičů bylo zapnuto filtrování podle údajů přihlášeného uživatele. Každému testu patří jeden uživatel, který je nastaven pomocí uživatelského

---

<sup>31</sup> Anotace `SlowTest`, obsahující tag „`slow_test`“.

## 4. TESTOVÁNÍ

---

```
1 mvc.perform(  
2     MockMvcRequestBuilders  
3         .post("/api/v1/comment")  
4         .contentType(MediaType.APPLICATION_JSON)  
5         .content(commentDTO.toString())  
6 )  
7 .andExpect(status().isCreated)  
8 .andExpect(content().contentType(MediaType.APPLICATION_JSON))  
9 .andExpect(jsonPath("$.createdById", is(  
10     commentDTO.createdById.toInt()  
11     )))  
12 .andExpect(jsonPath("$.text", is(commentDTO.text)))  
13 .andExpect(jsonPath("$.createdAt", is(  
14     commentDTO.createdAt.format(  
15         DateTimeFormatter.ISO_LOCAL_DATE_TIME  
16     )  
17     )))
```

---

Obrázek 4.5: Příklad použití nástroje MockMvc

jména<sup>32</sup>. V případě, že uživatel nemá dostatečná přístupová práva, řadič musí vrátit HTTP status s číslem 400<sup>33</sup>.

Současná testování obsahuje 106 integračních testů. Popis těchto testů je k nalezení v příloze D.2.

### 4.5 Samostatný profil pro testování

Testování aplikace vyžaduje vlastní nastavení, proto byl vytvořen samostatný profil, který odděluje testovací nastavení od nastavení produkce a vývoje. Profil má vlastní konfigurační soubor (`application-test.properties`), obsahující nastavení databáze.

Pro testování byla využita stejná databáze jako i pro produkci, za účelem kompletního testování aplikace za stejných podmínek. Takový postup zmenšuje šance na výskyt nečekaných chyb, působených použitím odlišné databáze v produkci. Automatické generování schématu bylo nastaveno na hodnotu `create-drop`, což znamená, že Hibernate obnoví schéma při nastartování `SessionFactory` a zruší schéma po zastavení `SessionFactory`.

---

<sup>32</sup>Uživatelským jménem v systému je adresa elektronické pošty.

<sup>33</sup>Bad Request.

Profil pro testování má též vlastní implementaci pro `UserDetailsService`<sup>34</sup>, obsahující tři implicitní uživatele:

- `root`, vlastníci role „ROLE\_USER“ a „ROLE\_ROOT“
- `userEmail`, vlastníci role „ROLE\_USER“
- `user2Email`, vlastníci role „ROLE\_USER“

Tyto implicitní uživatel jsou využity v integračních testech pro spouštění požadavků (`request`) přes instanci `MocMvc`. Podrobněji byly integrační testy popsány v sekci 4.4.

## 4.6 Pokrytí kódu testy

V této sekci bude popsáno pokrytí kódů testy a uvedeny statistiky. Za účelem zvýšení přesnosti provedené analýzy byly využity speciální nástroje pro zhodnocení pokrytí kódů.

### 4.6.1 JaCoCo

Prvním nástrojem, který byl využit pro analýzu testů, je JaCoCo. Tento nástroj byl podrobně popsán v sekci 1.5.3 . V této sekci je uveden pouze proces analýzy a její výsledky.

JaCoCo vyžaduje implementování vlastního úkolu<sup>35</sup>(`task`) v Gradle. Tento úkol byl nazván `jacocoTestReport` a je možné jej spustit přes příkazový řádek. Generování výsledků závisí na spouštění klasického úkolu `test` od Gradle.

Generování výsledků pokrytí testů bylo využito jako nápověda pro identifikování důležitého kódu nepokrytého testy. Proto byl úkol `jacocoTestReport` spuštěn několikrát. Podle poslední verze výsledků, která byla spuštěna pro poslední verzi aplikace, 82 % instrukcí je pokryté testy. Podrobné informace se můžete dozvědět v příloze D.

### 4.6.2 IntelliJ IDEA

Druhým testovacím nástrojem, který byl využit pro zjištění pokrytí kódů testy, je IntelliJ IDEA. Podrobněji byl tento nástroj popsán v sekci 1.5.4 .

Tento nástroj byl také využit jako pomocný nástroj při implementaci testů. V této sekci budou uvedeny jenom výsledky posledního spuštění pro poslední verzi aplikace. Kompletní informace o výsledcích jsou k nalezení v příloze D.

<sup>34</sup>Základní rozhraní, které framework Spring využívá pro načítání informací o uživateli.

<sup>35</sup>Konfigurovatelný úkol pro nástroj Gradle, například `test` nebo `build`.

#### 4. TESTOVÁNÍ

---

IntelliJ IDEA poskytuje nepatrně odlišnou informaci o pokrytí kódů než nástroj JaCoCo. Podle výsledků jsou pokryté testy:

- 79 % tříd
- 91,6 % metod
- 87,4 % řádek kódů

---

## Zhodnocení a budoucí kroky

V této kapitole bude zhodnocena použitelnost aplikace v době jejího odevzdání a navrženy vhodné budoucí kroky pro pokračování vývoje.

### 5.1 Zhodnocení výsledné aplikace

Tato sekce je věnována zhodnocení použitelnosti výsledného návrhu a implementace aplikace.

#### 5.1.1 Implementace

Výsledná implementace aplikace obsahuje všechny potřebné funkce pro správné fungování, a také byly splněny všechny požadavky frontendového týmu ohledně úprav návrhu (viz kapitolu 2). Byla též přidána podpora dodatečných požadavků pro následné propojení Android aplikace a serverového backendu (viz kapitolu 3). Výsledný stav API byl kontrolován autorem frontendové části aplikace Martinem Beranem. Provedená implementace v rámci této bakalářské práce je prodloužením již existující implementace. Proto pro provedené implementace s již existující implementací byla provedena analýza na základě počtu řádků kódu aplikace. Na obrázku 5.1 jsou zobrazeny výsledky analýzy. Pro analýzu byla uvažována jenom složka „be-springboot/src/main“. Stejná analýza byla provedena před začátkem práce pro již existující implementaci, která byla popsána v sekci 2.5.

Současná implementace aplikace je funkční a je aktuálně dostupná na adrese „<https://37.46.80.230:8778>“. Proces nasazování aplikace byl podrobně popsán v sekci 2.9. Dokumentace API aplikace je dostupná online na adrese „<https://37.46.80.230:8778/swagger-ui.html>“. Místo specifické IP adresy je také možné použít doménové jméno „rozvody.jagu.biz“. Výsledná verze aplikace je uložena do větve `dev` v rámci GitLab. Verze aplikace uložena do větve

## 5. ZHODNOCENÍ A BUDOUCÍ KROKY

---

```
236 text files.
236 unique files.
Complex regular subexpression recursion limit (32766) exceeded at /usr/local/Cellar/cloc/1.84/libexec/bin/cloc line 9879.
Complex regular subexpression recursion limit (32766) exceeded at /usr/local/Cellar/cloc/1.84/libexec/bin/cloc line 9879.
3 files ignored.

github.com/AlDanial/cloc v 1.84 T=0.21 s (1088.7 files/s, 42489.1 lines/s)
-----
Language          files      blank      comment      code
-----
Kotlin             232        1446         294         7339
JSON                1           0           0            14
-----
SUM:               233        1446         294         7353
-----
```

Obrázek 5.1: Analýza rozsahu provedené práce za účelem implementace programu

```
47 text files.
47 unique files.
2 files ignored.

github.com/AlDanial/cloc v 1.84 T=0.10 s (455.7 files/s, 72129.4 lines/s)
-----
Language          files      blank      comment      code
-----
Kotlin             38         463         276         6044
SQL                 7           88          12           239
-----
SUM:               45         551         288         6283
-----
```

Obrázek 5.2: Analýza rozsahu provedené práce za účelem testování programu

`master` není aktuální, protože současná aplikace ještě není kompletně připravená pro fungování v produkci.

Pro spuštění aplikace pomocí příkazového řádku je nutné zadat příkaz „./gradlew bootRun“ v kořenové složce. Další informace o podporovaných příkazech jsou uvedeny v souboru „README.md“, který se také nachází v kořenové složce.

### 5.1.2 Testování

Aplikace byla vhodně protestována pomocí unit testů a integračních testů (viz kapitolu 4). Analýza řádků kódů testování je na obrázku 5.2. Pro spuštění testů pomocí příkazového řádku je potřeba zadat příkaz „./gradlew test“ v kořenové složce aplikace. Proces běhu testů vyžaduje specificky nastavenou databázi PostgreSQL. Podrobný popis požadované konfigurace bude uveden v souboru „README.md“, který se také nachází v kořenové složce.

### 5.1.3 Bezpečnost

Předchozí verze aplikace obsahovala jenom návrh bezpečnosti pomocí přisvojení určitých rolí každému uživateli v rámci rodin. Tento návrh byl následně

implementován a rozšířen. Proces zjištění rolí uživatele vyžadoval implementaci procesu autorizace, proto byl použit protokol OAuth 2.0. Pro zaručení bezpečné komunikace mezi klientem a serverem byl také nahrazen protokol HTTP protokolem HTTPS.

## 5.2 Požadavky na změny

Pro zaručení správného návrhu API je potřeba propojit výsledný návrh serverového backendu a Android aplikace, která se řeší v rámci souběžné bakalářské práce. Proto je potřeba v rámci této práce připravit API, které by bylo použitelné. Současný návrh Android aplikace nepodporuje požadavky. Za účelem vyvarování kolizí výsledná implementace API serveru také nepodporuje požadavky na změny.

## 5.3 Návrh budoucích kroků

Tato sekce je věnovaná návrhu vhodných budoucích kroků pro pokračování vývoje aplikace.

### 5.3.1 Testování

Výsledný stav backendové části aplikace je připraven k použití současným stavem frontendové části aplikace. API serveru bylo kontrolováno autorem souběžně vyvíjenou frontendovou částí aplikace. Pro následující proces vývoje mají být nastavení Android aplikace upravena pro použití externího serverového backendu. Po provedených úpravách má být provedeno podrobné testování, jestli jsou všechny funkce implementovány správně a případně opraveny.

### 5.3.2 Implementace

Po úspěšném propojení serverového backendu a souběžně vyvíjené Android aplikace má být dokončena implementace chybějících funkcí. Do takových funkcí patří požadavky na změny v rámci rodiny.

### 5.3.3 Bezpečnost

V současné implementaci aplikace je autorizační server součástí serverového backendu. Proto, pro zvýšení bezpečnosti aplikace, je nutné přidat možnost přihlašování pomocí externího autorizačního serveru. Proces přihlašování se potom změní následujícím způsobem: klient se přihlásí do autorizačního serveru a obdrží *access token* a *refresh token*. Tyto *tokeny* klient potom může použít pro komunikaci se serverem. Server bude ověřovat platnost *tokenů* dotazováním na externí autorizační server.





---

## Závěr

Projekt byl rozdělen do dvou samostatných částí, které byly vyvíjeny paralelně. První částí je Android aplikace. Druhou částí, která je předmětem této bakalářské práce, je serverový backend, poskytující RESTové služby pro Android aplikace.

Cílem této práce bylo navržení vhodných úprav a následná implementace existujícího návrhu a fragmentů implementace. Také zhodnocení použitelnosti výsledné implementace a navržení vhodných budoucích kroků pro pokračování vývoje serveru. Při implementaci byl také uvažován současný stav souběžně vyvíjené frontendové části aplikace.

I když se autor této práce zúčastnil předmětů, během kterých byl udělán předchozí návrh aplikace a fragmenty implementace, byla potřeba přezkoumat návrh programu za účelem vylepšení návrhu a eliminování nalezených nedostatků během implementace frontendové a backendové části. Předchozí implementace programu pokrývala jenom malou část aplikace, proto skoro celá již existující implementace byla přepsána. Byl opraven doménový model, provedeny změny použité třívrstvé architektury, rozšířené API, rozšířená dokumentace API, přidán proces registrace a přihlašování uživatele, přidány další profily aplikace a také byla implementována bezpečnost aplikace na základě přihlášeného uživatele. Při implementaci byly uvažovány požadavky kolegy, který souběžně vyvíjí frontendovou část aplikace.

Důležitou částí této práce bylo testování, které je v rozsahu napsaného kódu porovnatelné s implementací serveru samotného. Testy byly rozděleny do dvou typů: unit testy a integrační testy. Unit testy testují samostatně testovatelné funkce programu. Integrační testy využívají za běhu celý kontext aplikace a testují, jestli jednotlivé aspekty aplikace fungují správně.

## ZÁVĚR

---

Výsledkem této bakalářské práce je funkční aplikace splňující všechny požadavky frontendové části aplikace. Ale, jak Android aplikace, tak i serverový backend ještě nejsou ve svém finálním stavu. V rámci této práce byl zhodnocen současný stav a představeny vhodné budoucí kroky pro pokračování vývoje.

---

## Seznam použité literatury

1. Contributing to Angular. In: *GitHub*. 2018. Dostupné také z: <https://github.com/angular/angular/blob/master/CONTRIBUTING.md>.
2. Oracle Java SE Support Roadmap. *Oracle*. Dostupné také z: <https://www.oracle.com/java/technologies/java-se-support-roadmap.html>.
3. KOTLIN FOUNDATION. Compatibility Tools. Dostupné také z: <https://kotlinlang.org/docs/reference/evolution/kotlin-evolution.html>.
4. KOTLIN FOUNDATION. Learn Kotlin. Dostupné také z: <https://kotlinlang.org/docs/reference/>.
5. GRADLE INC. Gradle vs Maven Comparison. © 2020. Dostupné také z: <https://gradle.org/maven-vs-gradle/>.
6. Ant vs Maven vs Gradle. *Bealdung*. Dostupné také z: <https://www.baeldung.com/ant-maven-gradle>.
7. Projects. *Spring*. © 2020. Dostupné také z: <https://spring.io/projects>.
8. Spring Framework. *Spring*. © 2020. Dostupné také z: <https://spring.io/projects/spring-framework>.
9. Externalized Configuration. *Spring Boot Reference Documentation*. © 2020. Dostupné také z: <https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-external-config>.
10. Spring Boot. *Spring*. © 2020. Dostupné také z: <https://spring.io/projects/spring-boot>.
11. Spring Security. *Spring*. © 2020. Dostupné také z: <https://spring.io/projects/spring-security>.

12. Query methods. *Spring Boot Reference Documentation*. © 2020. Dostupné také z: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods>.
13. Spring Web Services. *Spring*. © 2020. Dostupné také z: <https://spring.io/projects/spring-ws>.
14. Testing. *Spring Boot Reference Documentation*. © 2020. Dostupné také z: <https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html>.
15. BECHTOLD, Stefan; BRANNEN, Sam; LINK, Johannes; MERDES, Matthias; PHILIPP, Marc; RANCOURT, Juliette de; STEIN, Christian. JUnit 5 User Guide. Verze 5.6.2. © 2020. Dostupné také z: <https://junit.org/junit5/docs/current/user-guide/>.
16. Implementation Design. *JaCoCo - Java Code Coverage Library*. © 2009. Dostupné také z: <https://www.jacoco.org/jacoco/trunk/doc/implementation.html>.
17. KRISHNAN, Dilip; KELLY, Adrian. Springfox Reference Documentation. Verze 2.9.2-SNAPSHOT. © 2018. Dostupné také z: <https://springfox.github.io/springfox/docs/current/>.
18. H2. Tutorial. Dostupné také z: <http://www.h2database.com/html/tutorial.html>.
19. THE POSTGRES SQL GLOBAL DEVELOPMENT GROUP. PostgreSQL 12.3 Documentation. Dostupné také z: <https://www.postgresql.org/docs/12/index.html>.
20. AL, Danial. *CLOC. Version 1.84* [software]. 2020 [cit. 2020-05-17]. Dostupné z: <https://github.com/AlDanial/cloc>.
21. HTTP requests. *IBM Knowledge Center*. Dostupné také z: [https://www.ibm.com/support/knowledgecenter/SSGMCP\\_5.2.0/com.ibm.cics.ts.internet.doc/topics/dfht121.html](https://www.ibm.com/support/knowledgecenter/SSGMCP_5.2.0/com.ibm.cics.ts.internet.doc/topics/dfht121.html).
22. Profiles. *Spring Boot Reference Documentation* [online] [cit. 2020-05-17]. Dostupné z: <https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-profiles>.
24. Recurrence Rule. In: *icalendar.org* [online] [cit. 2020-05-11]. Dostupné z: <https://icalendar.org/iCalendar-RFC-5545/3-3-10-recurrence-rule.html>.
25. Task Execution and Scheduling. *Spring Boot Reference Documentation* [online] [cit. 2020-05-17]. Dostupné z: <https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-task-execution-scheduling>.

26. A Guide To Cron Expressions. *Bealdung* [online] [cit. 2020-05-17]. Dostupné z: <https://www.baeldung.com/cron-expressions>.
27. DEINUM, Marten. *Spring Boot 2 Recipes: A Problem-Solution Approach: Spring Task Scheduling*. Nizozemsko: Apress, 2018. ISBN 978-1-4842-3963-6.
28. About. In: *PostgreSQL.org* [online]. 2020 [cit. 2019-03-28]. Dostupné z: <https://www.postgresql.org/about/>.
29. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL 12.3 CREATE TYPE. Dostupné také z: <https://www.postgresql.org/docs/12/sql-createtype.html>.
30. PostgreSQL vs. MySQL. *PostgreSQL Tutorial* [online] [cit. 2020-05-17]. Dostupné z: <https://www.postgresqltutorial.com/postgresql-vs-mysql/>.
31. PostgreSQL vs MySQL. *2ndQuadrant* [online] [cit. 2020-05-17]. Dostupné z: <https://www.2ndquadrant.com/en/postgresql/postgresql-vs-mysql/>.
32. Annotation Type Tag. *JUnit 5.0.1 API* [online] [cit. 2020-05-17]. Dostupné z: <https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Tag.html>.
33. Integration Testing in Spring. *Bealdung* [online] [cit. 2020-05-17]. Dostupné z: <https://www.baeldung.com/integration-testing-in-spring>.



---

## Seznam doporučené literatury

23. GUTIERREZ, Felipe. *Pro Spring Boot 2 An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices*. USA: Apress, 2019. ISBN 978-1-4842-3676-5.





## Seznam použitých zkratk

**API** Application Programming Interface

**BI-SP1** Bakalářský předmět Softwarový týmový projekt 1 vyučovaný na FIT  
ČVUT

**BI-SP2** Bakalářský předmět Softwarový týmový projekt 2 vyučovaný na FIT  
ČVUT

**CLOC** Count Lines of Code

**ČVUT** České vysoké učení technické v Praze

**DSL** Domain-Specific Language

**DTO** Data Transfer Object

**FIT** Fakulta informačních technologií

**HTML** Hypertext Markup Language

**HTTP** Hypertext Transfer Protocol

**HTTPS** Hypertext Transfer Protocol Secure

**ID** Identifikátor

**IoC** Inversion of Control

**JVM** Java Virtual Machine

**REST** Representational State Transfer

**SQL** Structured Query Language

**URI** Uniform Resource Identifier



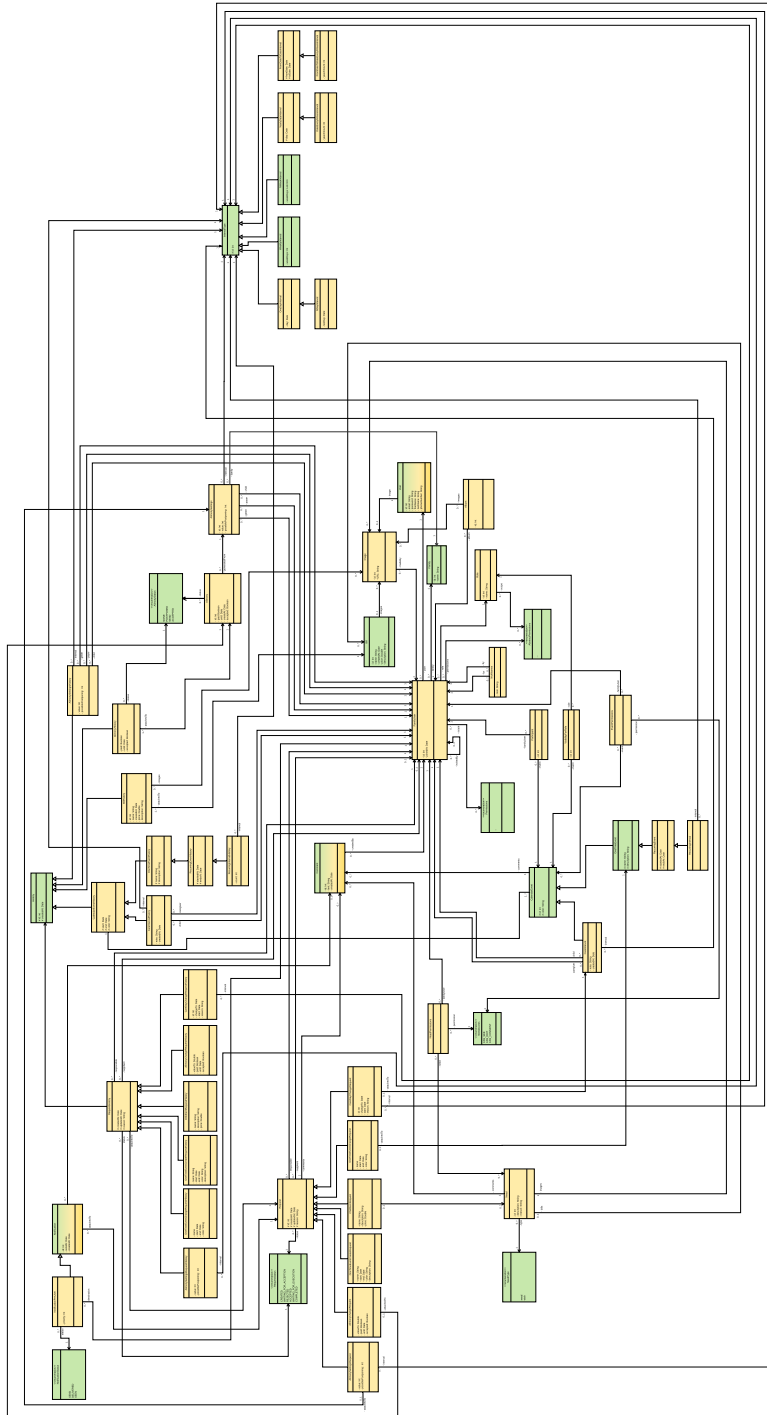
---

## Slovník pojmů

<b>Access token</b>	Token, který klient používá pro komunikaci se serverem.
<b>Backend</b>	Část aplikace, která se stará o data a jejich správu. Pro interakci s backendem klient většinou potřebuje přístup do frontendové části aplikace.
<b>CLOC</b>	Nástroj poskytující možnost spočítat počet řádků kódu v dané složce. Nástroj podporuje velký počet jazyků programování. Výsledek obsahuje počet řádků kódu oddělený od komentářů a prázdných řádků.
<b>Commit</b>	Proces, při kterém se uloží všechny změny provedené v rámci systému řízení verzí a zařadí se do historie změn.
<b>Datová vrstva</b>	Vrstva aplikace, která komunikuje s databází.
<b>DSL jazyk</b>	Programovací jazyk (nebo použití obecného programovacího jazyka) vytvořený za účelem vyřešení konkrétní problémové domény.
<b>Diagram aktivit</b>	Diagram aktivit zobrazuje, jak objekty spolupracují.
<b>Diagram Užítí</b>	Diagram Užítí popisuje chování systému z vnějšího pohledu.
<b>Doménový model</b>	Náčrt základních entit systému a vztahů mezi nimi.
<b>Drátový model</b>	Grafické zobrazení hlavních prvků frontendové části aplikace.

<b>Git</b>	Distribuovaný systém řízení verzí.
<b>GitHub</b>	Webová platforma pro vývoj softwaru pomocí systému řízení verzí Git.
<b>GitLab</b>	Webová platforma pro vývoj softwaru pomocí systému řízení verzí Git.
<b>Framework</b>	Softwarová struktura orientovaná na vyřešení problému nebo jeho zjednodušení při procesu vývoje softwarových projektu.
<b>Frontend</b>	Prezentační vrstva aplikace se kterou interaguje klient.
<b>Integrační testy</b>	Testy zaměřené na ověření správné komunikace mezi komponentami.
<b>Interní DSL jazyk</b>	DSL jazyk využívající obecný programovací jazyk.
<b>Inversion of Controle</b>	Princip, podle kterého kontrolu nad vytvořením a provázáním tříd vlastní framework.
<b>Kontext aplikace</b>	Pokročilý kontejner, který funguje podobně. <i>Bean-Factory</i> Načítá definice beanů, provazuje je a vydává v případě nutnosti.
<b>Mapování</b>	Proces přidání adresy konkrétnímu controlleru pomocí anotací frameworku Spring, která se zadává jako URI při odesílání požadavků na Server.
<b>Refresh token</b>	Token, který klient používá pro obnovení access tokenu po vypršení jeho platnosti.
<b>Token</b>	Alfanumerické heslo.
<b>Unit testy</b>	Testy zaměřené na ověření správnosti fungování samostatně testovatelné části programu.

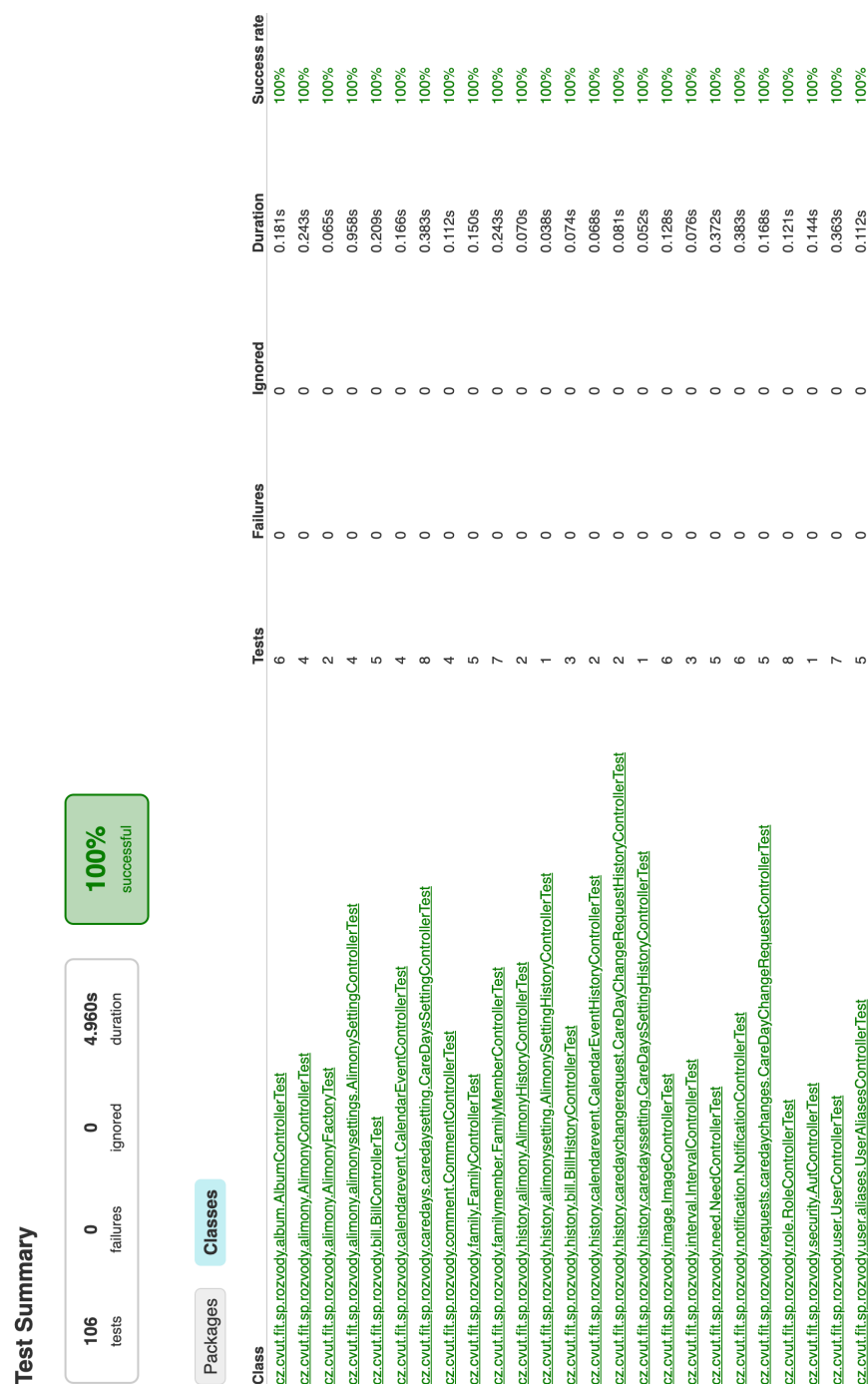
## **Doménový model před úpravami**



Obrázek C.1: Doménový model z předmětu BI-SP2

## **Testování**

## D. TESTOVÁNÍ



Obrázek D.1: Seznam tříd obsahujících unit testy vygenerovaný pomocí nástroje Gradle



## Test Summary

20 tests    0 failures    0 ignored    0.092s duration

100% successful

Packages

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
<a href="#">cz.cvut.fit.sp.rozvody.album.AlbumTest</a>	1	0	0	0s	100%
<a href="#">cz.cvut.fit.sp.rozvody.bill.BillTest</a>	1	0	0	0.024s	100%
<a href="#">cz.cvut.fit.sp.rozvody.image.ImageDIOTest</a>	2	0	0	0.002s	100%
<a href="#">cz.cvut.fit.sp.rozvody.image.ImageTest</a>	2	0	0	0.001s	100%
<a href="#">cz.cvut.fit.sp.rozvody.interval.IntervalTest</a>	5	0	0	0.045s	100%
<a href="#">cz.cvut.fit.sp.rozvody.recurrence.RuleTest</a>	9	0	0	0.020s	100%

Generated by Gradle 6.3 at 3 Jun 2020, 13:01:57

Obrázek D.2: Seznam tříd obsahující integračních testy vygenerovaný pomocí nástroje Gradle

## D. TESTOVÁNÍ

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
cz.cvut.fit.sp.rozvoxy.security		51%		25%	14	31	55	118	10	26	3	9
cz.cvut.fit.sp.rozvoxy.history.calendar		67%		22%	10	43	23	87	1	34	0	6
cz.cvut.fit.sp.rozvoxy.requests.caredaychanges		82%		70%	13	68	11	150	4	51	2	10
cz.cvut.fit.sp.rozvoxy.history.allimony		65%		50%	8	34	24	75	5	31	3	9
cz.cvut.fit.sp.rozvoxy.need		91%		63%	16	86	11	351	0	63	0	6
cz.cvut.fit.sp.rozvoxy.history.allimonysetting		64%		50%	7	33	35	86	6	32	5	11
cz.cvut.fit.sp.rozvoxy.history.bill		72%		50%	6	38	25	88	4	36	3	9
cz.cvut.fit.sp.rozvoxy.familymember		83%		56%	28	74	4	121	6	49	4	9
cz.cvut.fit.sp.rozvoxy.recurrence		89%		69%	35	100	2	142	1	38	0	7
cz.cvut.fit.sp.rozvoxy.family		80%		46%	19	52	6	100	6	38	3	8
cz.cvut.fit.sp.rozvoxy.bill		85%		57%	16	64	5	144	2	45	0	6
cz.cvut.fit.sp.rozvoxy.history.caredaychangerrequest		72%		25%	5	33	21	76	3	31	2	8
cz.cvut.fit.sp.rozvoxy.allimony		84%		73%	10	54	6	110	3	41	2	10
cz.cvut.fit.sp.rozvoxy.user		83%		50%	19	60	12	144	6	47	2	8
cz.cvut.fit.sp.rozvoxy.image		84%		63%	22	66	9	93	5	43	0	7
cz.cvut.fit.sp.rozvoxy.caredays.caredaysetting		85%		71%	11	55	8	93	4	41	1	8
cz.cvut.fit.sp.rozvoxy.caredays.caredaychange		76%		50%	11	35	5	57	7	29	4	8
cz.cvut.fit.sp.rozvoxy.history.caredaysetting		68%		50%	4	20	14	48	3	19	2	8
cz.cvut.fit.sp.rozvoxy.allimonysetting		87%		63%	15	58	12	131	4	43	4	11
cz.cvut.fit.sp.rozvoxy.caredays.careday		62%		50%	11	24	5	36	8	21	2	6
cz.cvut.fit.sp.rozvoxy.role		78%		50%	11	37	5	57	4	30	1	7
cz.cvut.fit.sp.rozvoxy.album		88%		68%	10	49	7	89	1	33	1	7
cz.cvut.fit.sp.rozvoxy.rest		48%		25%	6	15	16	29	4	13	0	3
cz.cvut.fit.sp.rozvoxy.caredays.caredayinterval		70%		50%	6	19	5	35	4	17	1	5
cz.cvut.fit.sp.rozvoxy.user.alias		89%		75%	7	50	4	105	0	36	0	6
cz.cvut.fit.sp.rozvoxy.comment		85%		78%	6	35	4	56	3	28	1	6
cz.cvut.fit.sp.rozvoxy.comment.parentcomment		79%		50%	5	22	4	43	3	20	1	4
cz.cvut.fit.sp.rozvoxy.interval		90%		75%	10	49	6	63	2	29	0	6
cz.cvut.fit.sp.rozvoxy.notification.alert		93%		78%	3	31	4	78	0	24	0	5
cz.cvut.fit.sp.rozvoxy.schedulejobs		62%		n/a	1	4	2	9	1	4	1	3
cz.cvut.fit.sp.rozvoxy.schedulejobs		61%		n/a	1	2	2	4	1	2	0	1
cz.cvut.fit.sp.rozvoxy.notification		96%		83%	1	19	1	19	0	16	0	4
cz.cvut.fit.sp.rozvoxy.need.enumerations		95%		n/a	2	6	0	10	2	6	0	2
cz.cvut.fit.sp.rozvoxy.requests.enumerations		96%		n/a	1	7	0	10	1	7	0	3
cz.cvut.fit.sp.rozvoxy.allimony.enumerations		95%		n/a	1	3	0	5	1	3	0	1
cz.cvut.fit.sp.rozvoxy.history		100%		n/a	0	13	0	13	0	13	0	3
Total	3,657 of 20,433	82%	282 of 742	61%	369	1,459	369	3,065	116	1,087	48	236

Obrázek D.3: Poslední verze výsledku pokrytí kódů testy podle JaCoCo

### Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	79% (192/ 243)	91.6% (987/ 1077)	87.4% (2707/ 3096)

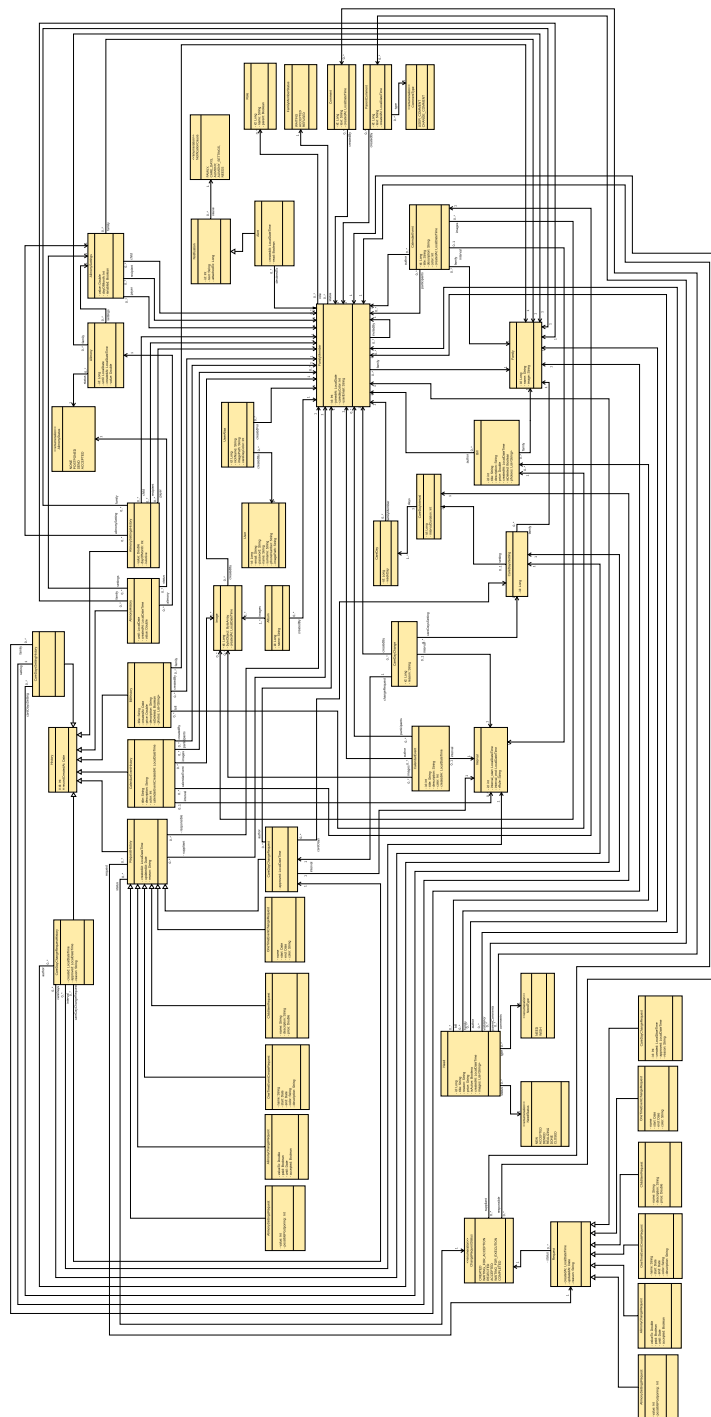
### Coverage Breakdown

Package ▲	Class, %	Method, %	Line, %
cz.cvut.fit.sp.rozvody	66.7% (2/ 3)	75% (3/ 4)	77.8% (7/ 9)
cz.cvut.fit.sp.rozvody.album	85.7% (6/ 7)	97% (32/ 33)	92.1% (82/ 89)
cz.cvut.fit.sp.rozvody.alimony	80% (8/ 10)	95% (38/ 40)	94.6% (105/ 111)
cz.cvut.fit.sp.rozvody.alimony.alimonysettings	63.6% (7/ 11)	90.7% (39/ 43)	91% (122/ 134)
cz.cvut.fit.sp.rozvody.alimony.enumerations	100% (1/ 1)	100% (2/ 2)	100% (5/ 5)
cz.cvut.fit.sp.rozvody.bill	100% (6/ 6)	100% (44/ 44)	96.5% (139/ 144)
cz.cvut.fit.sp.rozvody.calendarevent	100% (6/ 6)	97.9% (47/ 48)	90% (171/ 190)
cz.cvut.fit.sp.rozvody.caredays.careday	66.7% (4/ 6)	73.7% (14/ 19)	83.8% (31/ 37)
cz.cvut.fit.sp.rozvody.caredays.caredaychange	50% (4/ 8)	75.9% (22/ 29)	86.7% (52/ 60)
cz.cvut.fit.sp.rozvody.caredays.caredayinterval	80% (4/ 5)	76.5% (13/ 17)	85.7% (30/ 35)
cz.cvut.fit.sp.rozvody.caredays.caredaysetting	87.5% (7/ 8)	90.2% (37/ 41)	90.4% (85/ 94)
cz.cvut.fit.sp.rozvody.comment	83.3% (5/ 6)	96.3% (26/ 27)	92.9% (52/ 56)
cz.cvut.fit.sp.rozvody.comment.parentcomment	75% (3/ 4)	94.7% (18/ 19)	90.7% (39/ 43)
cz.cvut.fit.sp.rozvody.family	62.5% (5/ 8)	89.2% (33/ 37)	92.1% (93/ 101)
cz.cvut.fit.sp.rozvody.familymember	55.6% (5/ 9)	91.7% (44/ 48)	93.5% (116/ 124)
cz.cvut.fit.sp.rozvody.history	100% (3/ 3)	100% (13/ 13)	100% (13/ 13)
cz.cvut.fit.sp.rozvody.history.alimony	66.7% (6/ 9)	86.7% (26/ 30)	68% (51/ 75)
cz.cvut.fit.sp.rozvody.history.alimonysetting	54.5% (6/ 11)	81.2% (26/ 32)	59.3% (51/ 86)
cz.cvut.fit.sp.rozvody.history.bill	66.7% (6/ 9)	88.9% (32/ 36)	71.6% (63/ 88)
cz.cvut.fit.sp.rozvody.history.calendarevent	100% (6/ 6)	97.1% (33/ 34)	72.4% (63/ 87)
cz.cvut.fit.sp.rozvody.history.caredaychangerequest	75% (6/ 8)	90.3% (28/ 31)	72.4% (55/ 76)
cz.cvut.fit.sp.rozvody.history.caredayssetting	75% (6/ 8)	84.2% (16/ 19)	70.8% (34/ 48)
cz.cvut.fit.sp.rozvody.image	100% (7/ 7)	97.6% (40/ 41)	90.3% (84/ 93)
cz.cvut.fit.sp.rozvody.interval	100% (6/ 6)	93.1% (27/ 29)	90.8% (59/ 65)
cz.cvut.fit.sp.rozvody.need	100% (6/ 6)	100% (63/ 63)	96.6% (339/ 351)
cz.cvut.fit.sp.rozvody.need.enumerations	100% (2/ 2)	100% (4/ 4)	100% (10/ 10)
cz.cvut.fit.sp.rozvody.notification	100% (4/ 4)	100% (16/ 16)	94.7% (18/ 19)
cz.cvut.fit.sp.rozvody.notification.alert	100% (5/ 5)	100% (24/ 24)	94.9% (74/ 78)
cz.cvut.fit.sp.rozvody.recurrencerule	100% (7/ 7)	100% (37/ 37)	98.6% (142/ 144)
cz.cvut.fit.sp.rozvody.requests.caredaychanges	80% (8/ 10)	92.2% (47/ 51)	92.7% (139/ 150)
cz.cvut.fit.sp.rozvody.requests.enumerations	100% (3/ 3)	100% (6/ 6)	100% (10/ 10)
cz.cvut.fit.sp.rozvody.rest	100% (3/ 3)	69.2% (9/ 13)	44.8% (13/ 29)
cz.cvut.fit.sp.rozvody.role	87.5% (7/ 8)	96.8% (30/ 31)	96.6% (56/ 58)
cz.cvut.fit.sp.rozvody.schedulejobs	100% (1/ 1)	50% (1/ 2)	50% (2/ 4)
cz.cvut.fit.sp.rozvody.security	60% (9/ 15)	59.4% (19/ 32)	50.8% (66/ 130)
cz.cvut.fit.sp.rozvody.user	75% (6/ 8)	91.3% (42/ 46)	93.1% (135/ 145)
cz.cvut.fit.sp.rozvody.useraliases	100% (6/ 6)	100% (36/ 36)	96.2% (101/ 105)

Obrázek D.4: Poslední verze výsledku pokrytí kódů testy podle IntelliJ IDEA



## **Výsledný doménový model**



Obrázek E.1: Doménový model po navržení a implementaci všech úprav

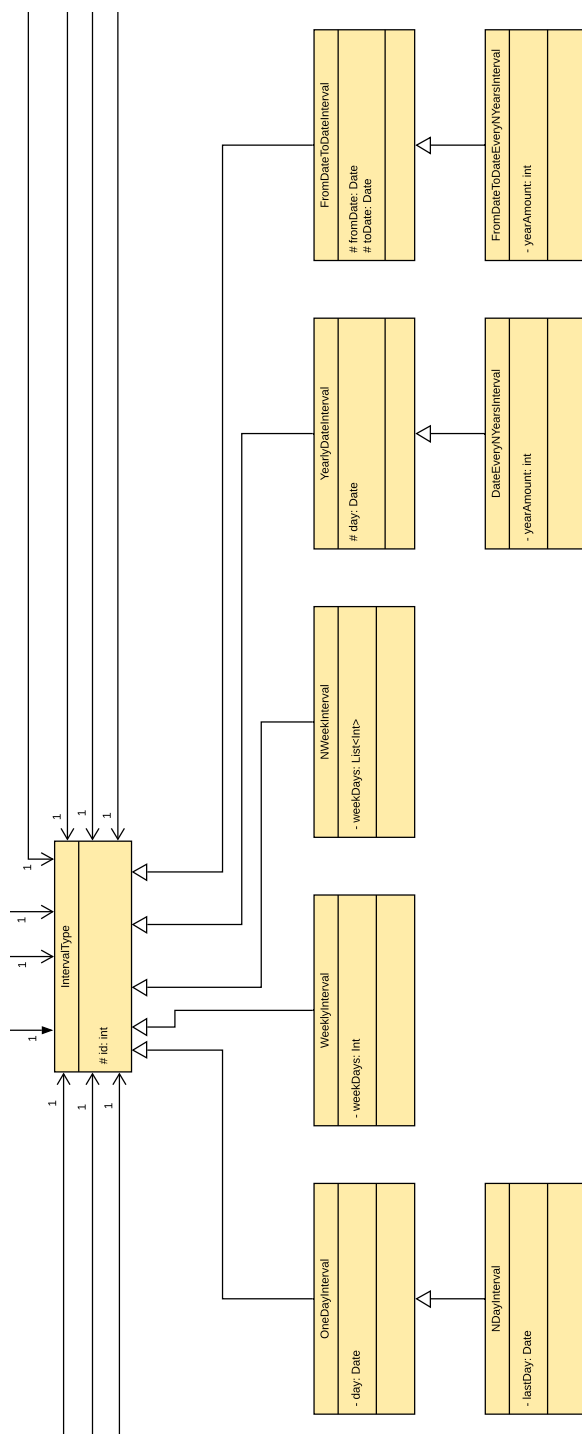
## **Konfigurace výsledného našeptávače pro chyby**

Typ chyby	HTTP status	zpráva	URL
<code>IllegalAccessException</code>	401	původní zpráva chyby	původní cesta
<code>IllegalArgumentException</code>	400	původní zpráva chyby	původní cesta
<code>NullPointerException</code>	500	nic	původní cesta
<code>No Such Element</code>	404	nic	nic
<code>MissingKotlinParameterException</code>	400	původní zpráva chyby	původní cesta

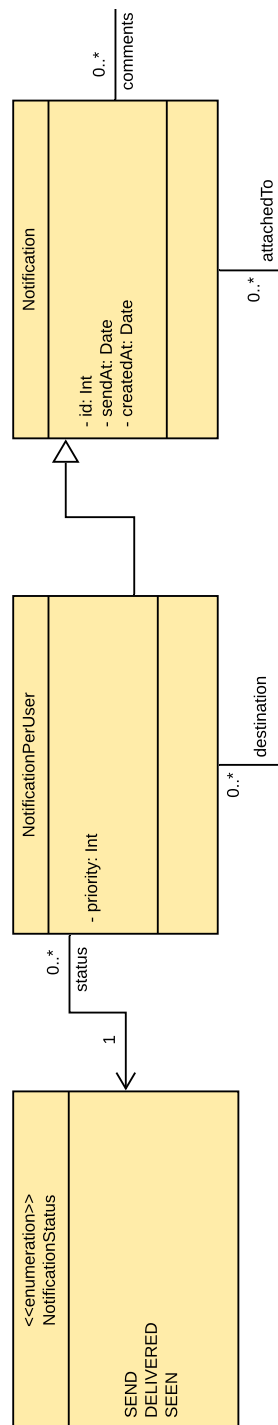
Obrázek F.1: Ukázka výsledné konfigurace našeptávače zachycování výjimek pro řadiče



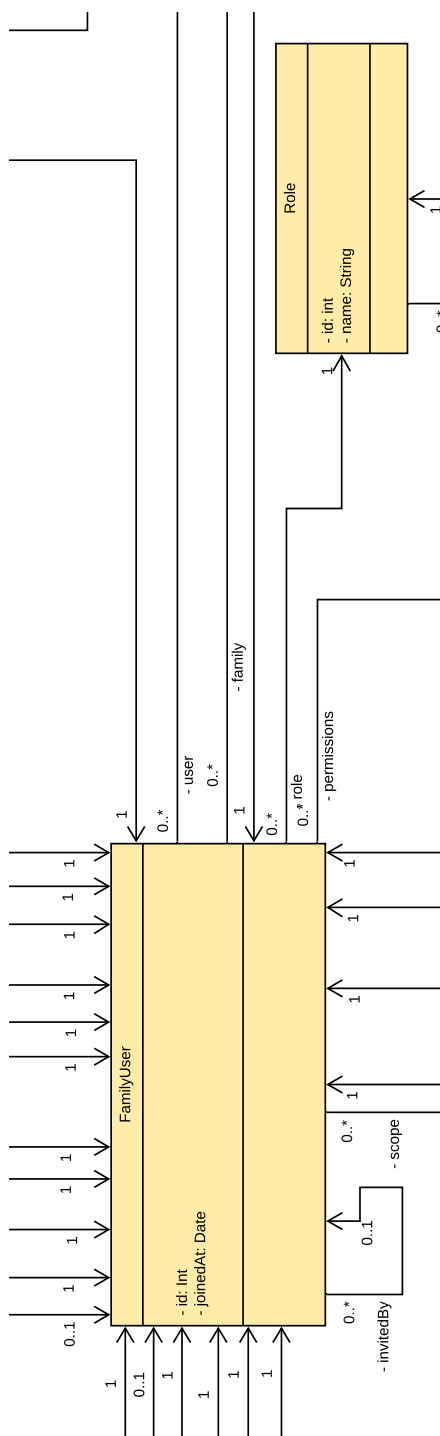
## Obrázky



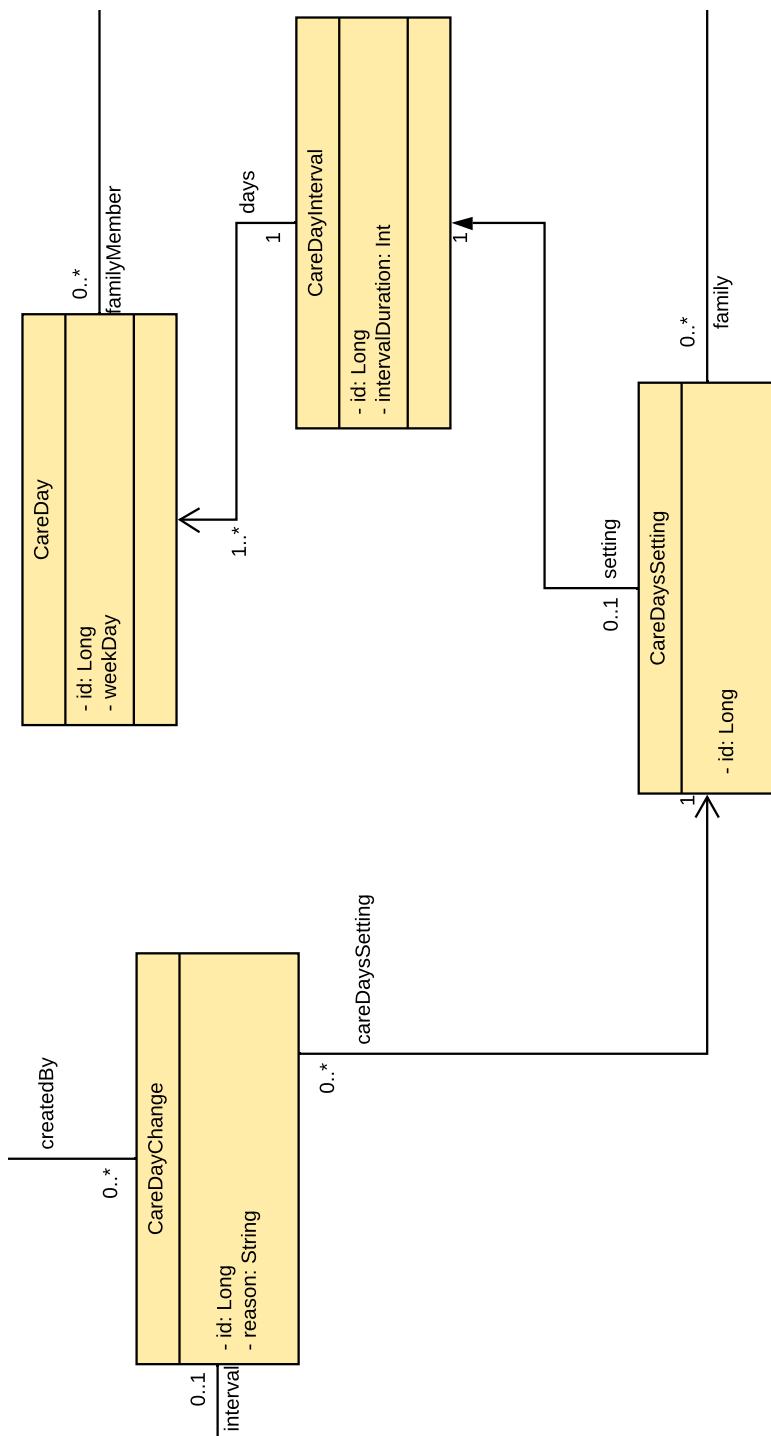
Obrázek G.1: Návrh entity *Interval* podle doménového modelu z předmětu BI-SP2



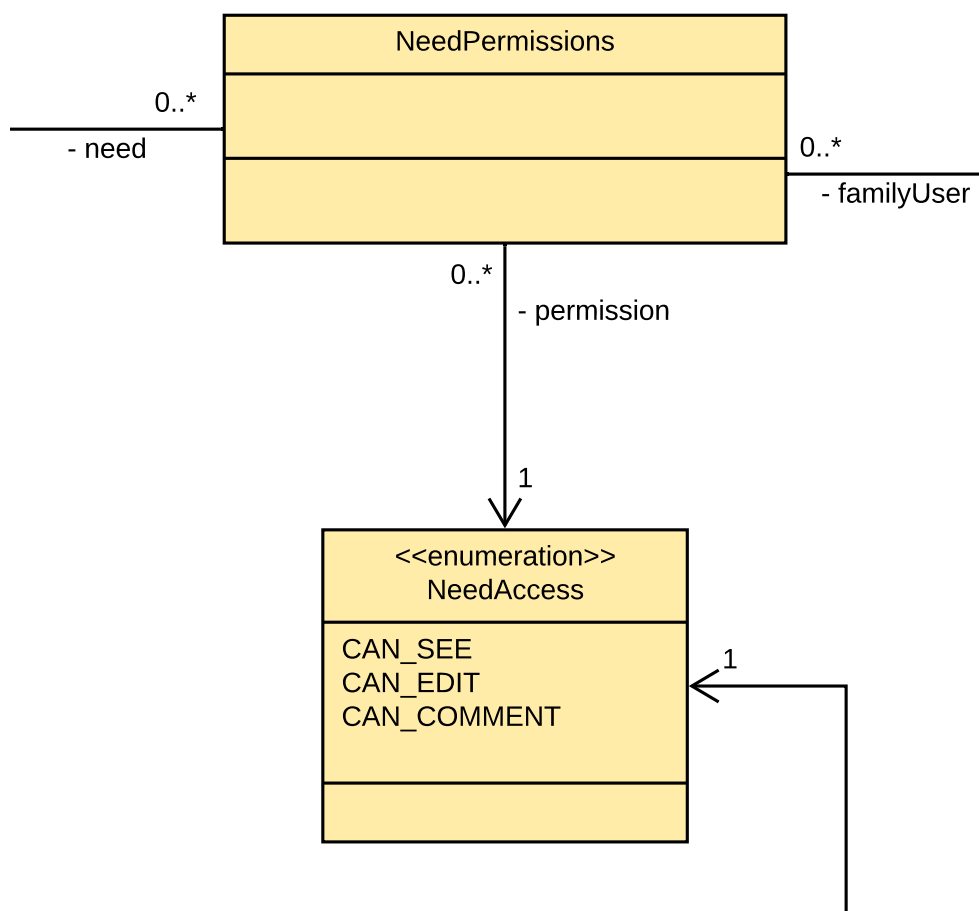
Obrázek G.2: Návrh oznámení podle doménového modelu předmětu BI-SP2



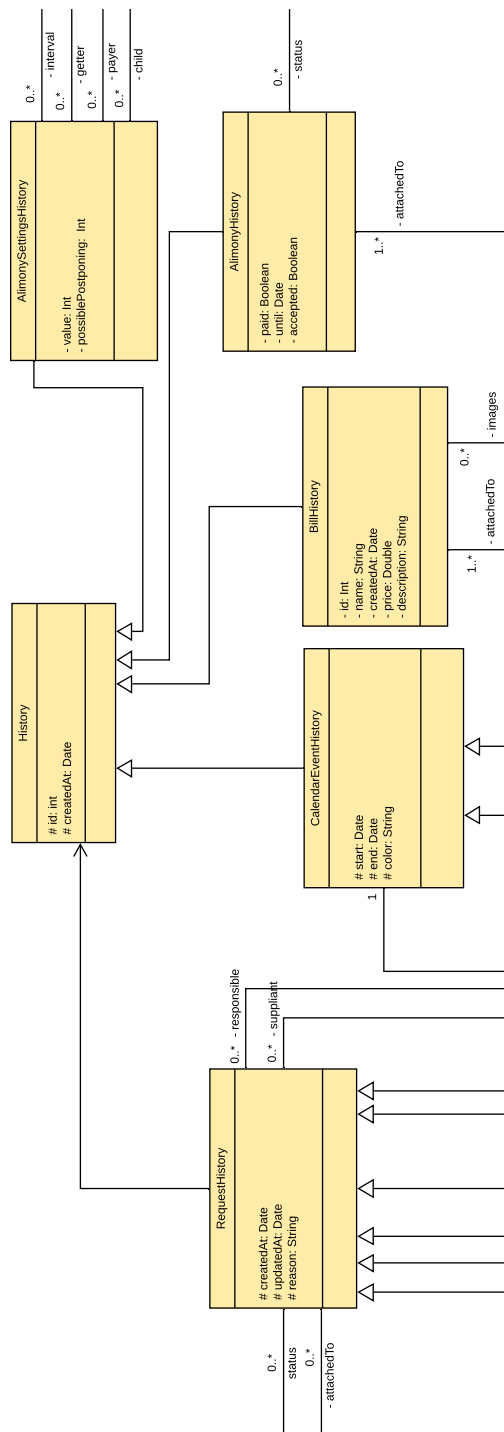
Obrázek G.3: Návrh entity Role podle doménového modelu z předmětu BI-SP2



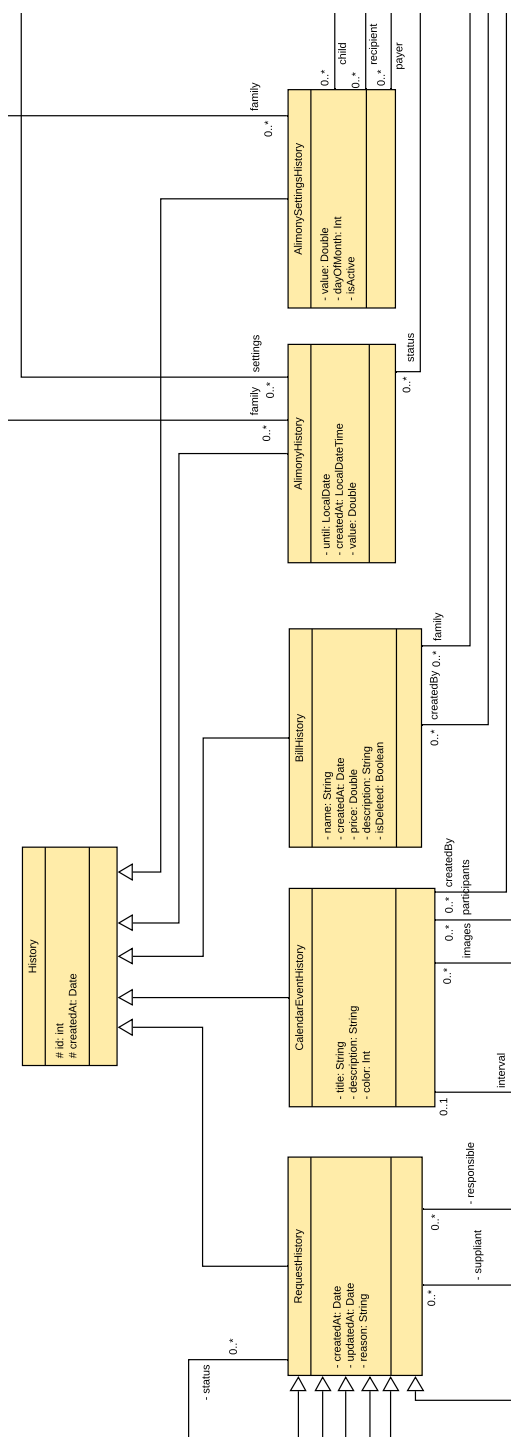
Obrázek G.4: Nový návrh pečovatelských dnů



Obrázek G.5: Návrh entity `NeedPermissions` podle doménového modelu z předmětu BI-SP2



Obrázek G.6: Předchozí návrh entity **History** podle doménového modelu z předmětu BI-SP2



Obrázek G.7: Návrh entity History po navržení změn pro entity, kterým patří příslušné entity historie



---

## Obsah přiloženého nosiče

README.md.....	stručný popis obsahu přiloženého nosiče
jar.....	adresář se spustitelnou formou implementace
src	
impl.....	zdrojové kódy implementace
thesis.....	zdrojová forma práce ve formátu $\text{\LaTeX}$
text.....	text práce
BP_Kolodka_Iaroslav_2020.pdf .....	text práce ve formátu PDF