Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

Studijní program: Aplikovaná informatika

Obor: Informační systémy a technologie

Porovnání frameworků Grails(Groovy) a Nette(PHP) pro rychlý vývoj webových aplikaci

DIPLOMOVÁ PRÁCE

Student: Jiří Frank

Vedoucí: Ing. Jarmila Pavlíčková

Oponent: Ing. Vladimír Oraný

2013

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze 1. května 2013 . . . . . .

Jméno a příjmení studenta

Poděkování

Tímto bych rád poděkoval vedoucí práce Ing. Jarmile Pavlíčkové za velkou ochotu, vstřícnost, cenné rady a připomínky.

Abstrakt

Tato práce se zaměřuje na srovnání frameworků pro rychlý vývoj aplikací Nette a Grails. Tohoto hlavního cíle je dosaženo stručným popisem obou jazyků (PHP, Groovy) a frameworků. Dále pak definováním poměrně komplexní sady kritérií pro samotné srovnání. Na základě těchto kritérií jsou pak oba frameworky porovnávány. Poslední částí je vývoj identické aplikace v obou prostředích, díky kterému je možné učinit srovnání i z ryze praktického hlediska. Přínos práce spočívá jednoznačně v poměrně zevrubném porovnání platforem Nette (PHP) a Grails (Groovy), které může posloužit jako základ pro rozhodování, jakou platformu pro vývoj nové aplikace zvolit. Srovnání jako takové může být využito i jako informace o tom, na co si dát při vývoji pozor, co programátorovi usnadní práci či naopak znepříjemní nebo znesnadní život, zejména v kontrastu s tím, co nabízí konkurenční platforma.

Klíčová slova

porovnání, framework ,Grails, Groovy, PHP, Nette, dynamický jazyk, RAD, kritéria

Abstract

This thesis aims on comparison of rapid development frameworks Nette and Grails. Main objective of this paper is achieved by brief description of both languages (PHP, Groovy) and frameworks. Next is defined relatively complex set of criteria for comparison itself. Frameworks are then compared based on this set. Last part is development of the identical application on both platforms, thanks to which the practicle comparison may be done. The main contribution of this work is in comprehensive comparison of platforms Nette (PHP) and Grails (Groovy), which may be used for making decision about which platform to take for a development of new application. Comparision of course may serve like an information about that what developer should keep an eye on, what can make life easier or harder especially in contrast with that what offers concurent platform.

Keywords

comparison, framework, Grails, Groovy, PHP, Nette, dynamic language, criteria, RAD

Obsah

1 Úvod ... 1
1.1 Cíl práce ... 1
1.2 Důvody výběru tématu a platforem ... 2
1.3 Předpoklady a omezení práce ... 2
1.4 Okruh čtenářů ... 2
1.5 Struktura práce ... 2
1.6 Přínosy práce ... 3
1.7 Existující podobné práce ... 3
1.8 Terminologie ... 4
1.8.1 Framework ... 4
1.8.2 Webový framework pro rychlý vývoj aplikací (RAD framework) ... 5
1.8.3 Dynamický a statický jazyk ... 6
1.8.4 Dynamické a statické typování ... 6
1.8.5 Interpretované a kompilované jazyky ... 7
1.8.6 Návrhový vzor Model-View-Presenter a Model-View-Controller ... 7
1.8.7 Platforma ... 9
2 Vybrané platformy pro vývoj ... 10
2.1 Dynamické jazyky ... 10
2.1.1 Popularita dynamických jazyků ... 11
2.2 Výběr frameworků ... 12
2.3 Stručná historie a charakteristika PHP ... 12
2.3.1 Charakteristické rysy ... 14
2.3.2 Framework Nette ... 15
2.4 Stručná historie a charakteristika Groovy ... 16
2.4.1 Charakteristický rysy ... 17
2.4.2 Framework Grails ... 18
3 Kritéria pro srovnání ... 20
3.1 Dostupnost ... 20
3.2 Podpora ... 21
3.3 Architektura frameworku ... 22
3.4 Práce s adresami ... 22
3.5 Convention over Configuration ... 24
3.6 Dependency Injection ... 25
3.7 Podpora testování ... 27
3.8 Integrované funkce v základu ... 29

3.9 Dostupné běhové prostředí ... 30
3.10 AJAX/AJAJ ... 32
3.11 Persistentní a doménová vrstva ... 33
3.12 Šablonování ... 37
3.13 Možnosti rozšíření ... 39
3.14 Podpora lokalizací ... 41
3.15 Generátory ... 42
3.16 Rozšířenost a význam ... 44
3.17 Integrované vývojové prostředí ... 46
3.18 Hodnocení z praktického vývoje ... 47
3.19 Aktivní vývoj ... 49
3.20 Komunita ... 50
3.21 Dokumentace ... 52
3.22 Pracovní nabídky ... 53
3.23 Srovnání frameworků ... 55
4 Návrh aplikace ... 58
4.1 Popis aplikace ... 58
4.1.1 Účel aplikace ... 58
4.1.2 Funkční specifikace ... 59
4.1.3 Předpoklady a omezení ... 59
4.2 Návrh aplikace ... 59
4.3 Globální návrh aplikace ... 60
4.3.1 Popis případů užití ... 60
4.3.2 Návrh tříd a jejich odpovědností ... 63
4.3.3 Vytvoření statického diagramu tříd ... 64
4.3.4 Systémový návrh ... 65
4.3.5 Návrh uživatelského rozhraní ... 66
4.4 Detailní návrh aplikace ... 68
5 Vývoj ... 72
5.1 Instalace a zprovoznění ... 72
5.1.1 Framework Nette ... 72
5.1.2 Framework Grails ... 73
5.1.3 Srovnání ... 74
5.2 Adresářová struktura a umístění důležitých souborů ... 74
5.2.1 Framework Nette ... 74
5.2.2 Framework Grails ... 75
5.2.3 Srovnání ... 76

5.3 Nastavení aplikace ... 77
5.3.1 Framework Nette ... 77
5.3.2 Framework Grails ... 79
5.3.3 Srovnání ... 80
5.4 Implementace modelu ... 80
5.4.1 Framework Nette ... 81
5.4.2 Framework Grails ... 83
5.4.3 Srovnání ... 84
5.5 Implementace controllerů/presenterů ... 85
5.5.1 Framework Nette ... 85
5.5.2 Framework Grails ... 88
5.5.3 Srovnání ... 90
5.6 Implementace view ... 90
5.6.1 Framework Nette ... 91
5.6.2 Framework Grails ... 92
5.6.3 Srovnání ... 95
5.7 Implementace ostatních tříd ... 95
5.7.1 Framework Nette ... 96
5.7.2 Framework Grails ... 96
5.8 Implementace testů ... 96
5.8.1 Framework Nette ... 96
5.8.2 Framework Grails ... 98
5.8.3 Srovnání ... 99
5.9 Zhodnocení vývoje u obou frameworků ... 99
6 Závěr ... 101
7 Terminologický slovník ... 103
8 Použité zdroje ... 107
9 Seznamy ... 112
9.1 Tabulky ... 112
9.2 Obrázky ... 113
9.3 Zdrojové kódy ... 114
10 Příloha 1 – Případy užití ... 116
11 Příloha 2 - Stanovení vah jednotlivých kritérií ... 122
12 Příloha 3 – Zdrojové kódy vytvořených aplikací ... 123

1 Úvod

V posledních letech jsou velmi znatelné tendence přechodu firem k vývoji primárně webových aplikací a ústupu od aplikací desktopových. Důvody popisuje dobře Ryan Metzner [Metzner, 2012], stačí vytvořit jedinou aplikaci, kterou budou moci uživatelé spustit v podstatě na všech platformách a zařízeních bez nutnosti nákladné portace. Tyto aplikace jsou narozdíl od běžného softwaru optimalizované pro masový veřejný provoz na internetu (jejich obsah je v případě kvalitně napsaného kódu indexovatelný a tudíž i vyhledatelný populárními webovými vyhledávači). Na druhou stranu je nutné zmínit, že i když prodělaly webové aplikace velmi zajímavý rozvoj, stále nedosahují v množství funkcionalit a komfortu práce kvalit desktopového software [Marc, 2012].

Díky výše uvedenému trendu si získaly webové technologie, resp. aplikace, značnou přízeň vývojářů a podstatně vzrostly komunity zabývající se programovacími jazyky určenými výhradně, nebo částečně pro tvorbu webových aplikací. [Tiobe Software, 2013] Jako příklad bych zmínil jazyky PHP, Java, Ruby, Groovy nebo C#. Spolu s tlakem na rychlý vývoj aplikací a vznikem agilních metodik vývoje začaly vznikat tzv. webové frameworky pro rychlý vývoj aplikací, které zjednodušily vývojářům práci vyřešením často se opakujících „problémů“.

Tato práce je zaměřena na porovnání frameworků Nette (jazyk PHP) a Grails (primárně jazyk Groovy).

1.1 Cíl práce

Cílem práce je velice stručně popsat framework Nette včetně jazyka PHP a framework Grails, včetně jazyka Groovy a následně definovat ucelenou sadu kritérií pro porovnávání frameworků určených pro rychlý vývoj aplikací. Poté na základě této sady frameworky porovnat. Dále pak pro ilustrování rozdílů při praktickém vývoji vyvinout aplikaci na obou těchto platformách. Předpokladem pro vývoj je přiměřeně rozsáhlý návrh aplikace.

1.2 Důvody výběru tématu a platforem

Toto téma jsem si vybral především proto, že tématika webových frameworků je velmi aktuální a velice zajímavá. Navíc porovnání českého Nette a Grails zatím nebylo publikováno i přes to, že jsou v podstatě přímí rivalové.

Nette je jedním z nejpoužívanějších a nejpopulárnějších PHP frameworků v České republice s velmi bohatou a aktivní komunitou (naneštěstí toto tvrzení nelze reálně podložit, nicméně komunita, která se shromáždila okolo Nette, tuto domněnku potvrzuje). Dalším neméně podstatným důvodem je to, že mám s tímto frameworkem zkušenosti z komerčního vývoje.

Dalším porovnávaným frameworkem je Grails, který je více populární v zahraničí, než v České republice. I přes tento fakt a rozdílnost jazyků ho však považuji za velice zajímavou alternativu k Nette a tudíž i pro samotné porovnání.

1.3 Předpoklady a omezení práce

Práce předpokládá u čtenáře základy programování, ideálně pak zkušenosti s Groovy a PHP. Zároveň je nutné brát v úvahu, že cílem práce není být dobrou příručkou či návodem k vývoji na daných platformách, ale charakter textu je spíše srovnávací a hodnotící.

1.4 Okruh čtenářů

Práce je určena především programátorům, zejména pak těm, kteří mají zkušenosti s některým z frameworků a rádi by si přečetli o možnostech, které nabízí alternativní platforma nebo se dozvěděli o výhodách a nevýhodách jednotlivých přístupů. Tato práce může být zajímavá i pro architekty či analytiky, kteří zde mohou nalézt informace o silných a slabých stránkách, nebo lépe pochopit, zda se pro daný projekt lépe hodí ta či ona platforma.

Nesporně zajímavá může být práce i pro začínající programátory, kteří se rozhodují, jaký framework pro rychlý vývoj webových aplikací zvolit.

1.5 Struktura práce

Celá diplomová práce je rozdělena do tří hlavních částí. První část obsahuje stručný popis jazyků a frameworků, včetně krátkého shrnutí historického vývoje. Zde se čtenář

seznámí s tím, jaké technologie budu srovnávat. V další části se budu věnovat definování kritérií pro srovnání webových frameworků pro rychlý vývoj aplikací. Na základě takto definované sady porovnám dva zmiňované frameworky. Poslední část práce bude ryze praktická, proběhne zde nejprve návrh vzorové aplikace a následně i její vývoj v obou prostředích.

1.6 Přínosy práce

Přínos práce spočívá jednoznačně v poměrně podrobném srovnání frameworků, resp. platforem Nette (PHP) a Grails (Groovy), které může posloužit jako základ pro rozhodování, jaký framework pro vývoj nové aplikace zvolit. Srovnání proběhne na základě definované sady kritérií, která je dostatečně komplexní i pro obecné porovnávání frameworků pro rychlý vývoj aplikací. Srovnání jako takové může být použito i jako informace o tom, na co si dát při vývoji pozor, co programátorovi usnadní práci či naopak znepříjemní či znesnadní, zejména v kontrastu s tím, co nabízí konkurenční platforma.

1.7 Existující podobné práce

Zde bych rád poukázal a stručně uvedl některé z prací, kde se již autoři podobným srovnáním, jako si klade za cíl tato práce, zabývali. První porovnání, na které jsem narazil, provedl [Odehnal, 2011]. Jedná se o porovnání frameworků Zend a Grails, při kterém se autor zabývá komplexním srovnáním obou platforem se zaměřením na návrhové, architektonické vzory a dojmy z praktického vývoje.

Další srovnání provedl [Hanyš, 2010], který se zabýval porovnáním javascriptových frameworků jQuery, MooTools a dalšími. V práci nejprve objasnil terminologii, provedl stručný popis frameworků. Posléze provedl velice stručné porovnání na základě několika málo kritérií.

Srovnáním webových frameworků pro rychlý vývoj aplikací na platformě Java se zabýval v minulosti [Zilvar, 2010]. Ve své práci srovnal framework Grails a Tapestry. Nejprve popsal oba frameworky a jejich principy, posléze definoval množinu kritérií, na jejichž základě provedl samotné srovnání. Díky tomu, že v dané době byly frameworky poměrně mladé a velice dynamicky se rozvíjející, určitý význam měla pravděpodobně i rešerše cizojazyčných zdrojů.

[Hradecký, 2009] provedl rozsáhlejší srovnání několika softwarových frameworků (Django, Cocoon, Tapesty, Symfony…). V této práci byla postupně definována objektivní i subjektivní kritéria pro srovnání a na základě těchto a praktického vývoje diskuzního fóra byly tyto frameworky porovnány. Přínosem je jednoznačně větší počet srovnávaných frameworků najednou.

V další práci při srovnání webových aplikačních frameworků pro rychlý vývoj aplikací postupoval [Válek, 2008] nejprve tak, že specifikoval vlastnosti frameworku CakePHP a Ruby on Rails. V praktické části práce se zaměřil na vývoj referenční aplikace v obou prostředích s následným vyhodnocením poznatků.

Podobným postupem jako výše uvedené se ubírá i [Florian, 2011], který srovnává framework Django a Ruby on Rails. Opět nejprve specifikuje oba frameworky, tedy jejich klíčové vlastnosti a přístupy. Poté definuje několik kritérií, podle kterých následně oba frameworky porovnáná.

Srovnáním Nette a Zend, dvou velice dobrých frameworků v jazyce PHP, se zabýval [Olišar, 2012]. Ve své práci práci definuje nejprve obecnou terminologii, přístupy a návrhové vzory. Posléze definuje sadu kritérií, podle kterých porovnává oba frameworky na základě zkušeností získaných z vývoje vzorové aplikace v obou prostředích.

1.8 Terminologie

Tato podkapitola si klade za cíl stručně vysvětlit některé pojmy, které považuji v této práci za důležité pro pochopení následujícího textu. Většina zkratek a nejednoznačně pochopitelných názvů bude však uvedena v terminologickém slovníku.

1.8.1 Framework

Definic slova framework existuje celá řada a každá se dívá na význam svým způsobem. Poměrně výstižně definoval framework [Clifton, 2003], který identifikoval jeho vlastnosti ve třech perspektivách:

a) Obálka[1]
    a. Zjednodušuje přístup k technologii
    b. Redukuje/eliminuje opakované úlohy

[1] Ve smyslu wrapperu

c. Zvyšuje flexibilitu aplikace pomocí abstrakce
d. Jsou často znovu použitelné bez složitých úvah o designu

b) Architektura
a. Spravuje kolekci objektů
b. Implementuje návrhové vzory

c) Metodologie
a. Vynucuje dodržování jednotného designu aplikace
b. Dekomponuje závislosti
c. Je často použitelný bez ohledu na typ vyvíjené aplikace

Další z výkladů definuje například JavaFramework[2]:

> *„Framework je sada předpřipravených stavebních bloků, které mohou programátoři použít, rozšířit nebo přizpůsobit specifickým potřebám. Díky frameworku nemusí vývojáři začínat od nuly pokaždé, když začínají psát novou aplikaci. "*
>
> [Přeloženo z originálu]

Wikipedia říká:

> *„Framework je softwarová struktura, která slouží jako podpora při programování a vývoji a organizaci jiných softwarových projektů. Může obsahovat podpůrné programy, knihovny API, podporu pro návrhové vzory nebo doporučené postupy při vývoji."*

Všechny tři definice dobře přibližují význam slova framework s názorným příkladem odlišného výkladu dle konkrétního úhlu pohledu daného autora.

1.8.2 Webový framework pro rychlý vývoj aplikací (RAD framework)

Jedná se o specifičtější termín, než je framework a významově z něj vychází. Je poměrně obtížné význam přesně vydefinovat, ale v případě, že bych měl velice stručně shrnout společné rysy webových RAD frameworků (Grails, Nette, CakePHP, Rails a další), definoval bych je takto:

a) Dynamický jazyk

[2] Citát pochází z oficiálních stránek JavaFrameworku, který je k dispozici na adrese http://www.jfwk.com/what_is.html

b) MVC[3]
c) Routování
d) Persistentní vrstva
e) Cachování
f) Šablonovací systém
g) Podpora testování

Každý z těchto pojmů bude detailněji rozebrán v dalších kapitolách. Dle [Towards the Rapid Application Development, 2011] jsou tyto frameworky vhodné zejména pro podporu vývoje často se opakujících standardních úkolů.

1.8.3 Dynamický a statický jazyk

> *„Dynamické jazyky mají schopnost měnit program při spuštění, zahrnujíc změny struktury objektů, typů či chování. Dynamické jazyky vám dovolují dělat takové věci při spuštění, které statické jazyky dělají při kompilaci, dovolují vám spouštět funkce, které byly vytvořeny za běhu.“*

Přeloženo z [Subramaniam, 2008]

Výše citovaná vlastnost velice zásadně mění pole možností, které má vývojář, resp. program k dispozici. Příkladem dynamického jazyka může být právě Groovy a u statického jazyka Java. Dynamickým jazykům se budu detailněji věnovat v další kapitole.

1.8.4 Dynamické a statické typování

Rozdíl mezi dynamickým a statickým typováním lze definovat zjednodušeně tak, že v případě staticky typovaného jazyka je nutné definovat konkrétní datové typy (jinak se program díky typové kontrole nepřeloží), u dynamicky typovaného jazyka dochází k určení typu až při spuštění programu. [Subramaniam, 2008]

Dále je možné typování rozlišit na slabé a silné. Silně typované jazyky (Java) provádí při operacích kontrolu datového typu, na rozdíl od slabě typovaných jazyků (C), které se bez jakýchkoliv problémů pokusí takovou operaci provést i bez kontroly. Pro ilustraci, sčítání řetězce a číselného datového typu je nesprávné, ale slabě typované jazyky tuto operaci provedou a vrátí třeba i nesmyslný výsledek. [Křivánek, 2004]

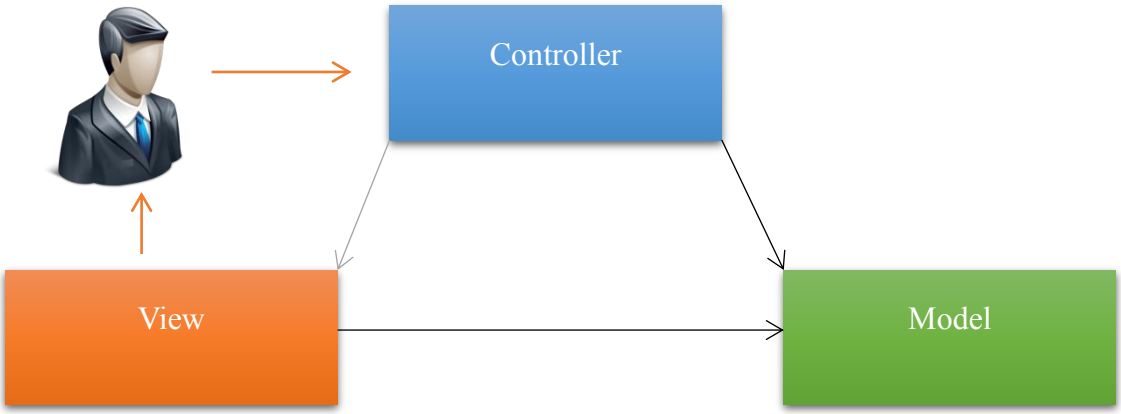
[3] Návrhový vzor model-view-controller

1.8.5 Interpretované a kompilované jazyky

Programovací jazyky lze rozlišovat různým způsobem. Jedním z nich je dělení na interpretované (PHP, Ruby…) a kompilované (C, C#...). Zjednodušeně řečeno, hlavní rozdíl spočívá v tom, že kompilovaný jazyk musí být nejprve přeložen do strojového či byte kódu, zpravidla bývá vygenerován nějaký přímo nebo nepřímo spustitelný soubor. Oproti tomu interpretované se nepřekládají předem, ale přímo za běhu tzv. interpreterem, který převádí jazyk do strojového kódu. V praxi se lze často setkat s kombinovaným přístupem (Java). Každý z typů má své výhody a nevýhody, jejichž vysvětlování je však mimo rozsah této práce. [Pierce, 2002]

1.8.6 Návrhový vzor Model-View-Presenter a Model-View-Controller

Model-View-Controller, zkráceně MVC, je architektonický vzor používaný při vývoji aplikací. Jedná se o poměrně starý vzor (vznikl již v sedmdesátých letech). Myšlenka tohoto vzoru spočívá v dekompozici aplikace do tří částí dle účelu, a to na zobrazovací (View), logickou (Controller) a datovou (Model). Tudíž dojde k oddělení vazeb mezi jednotlivými logickými celky v ideálním případě tak, aby změna jednoho příliš neovlivnila další části. Implementace toho návrhového vzoru přinese výhody v přehlednějším výsledném kódu, snazší kooperaci expertů v rámci týmu a jednodušší údržbě. Controller v tomto případě zodpovídá za logiku aplikace a manipulaco s Modelem. Vztah s View je slabší. Funguje zde jako obsluha a rozhoduje, které View se má použít a co se mu má předat. S Modelem pak pracuje View již samo. Model obstarává business logiku aplikace. Architekturu ilustruje Obrázek 1. [Bernard, 2009]

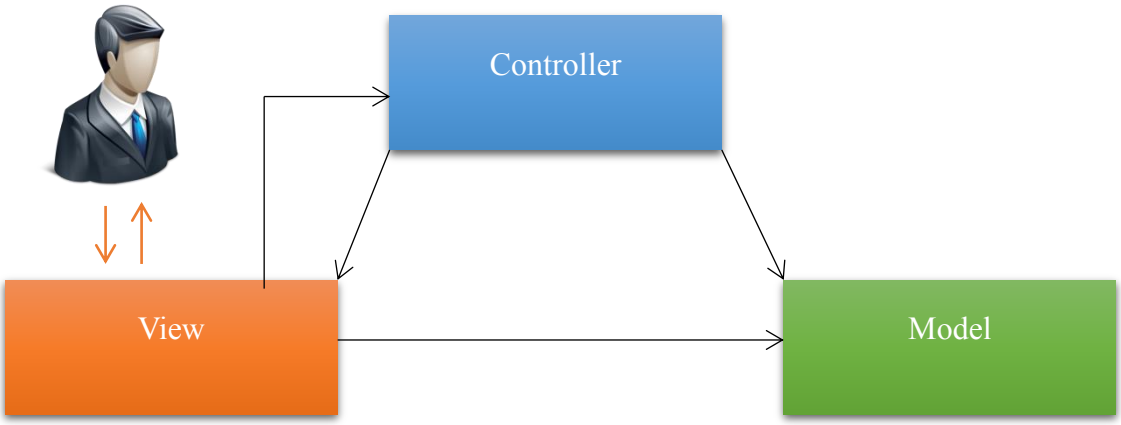


Obrázek 1 – Jedna z možných implementací MVC. Zdroj: Autor dle [Bernard, 2009]

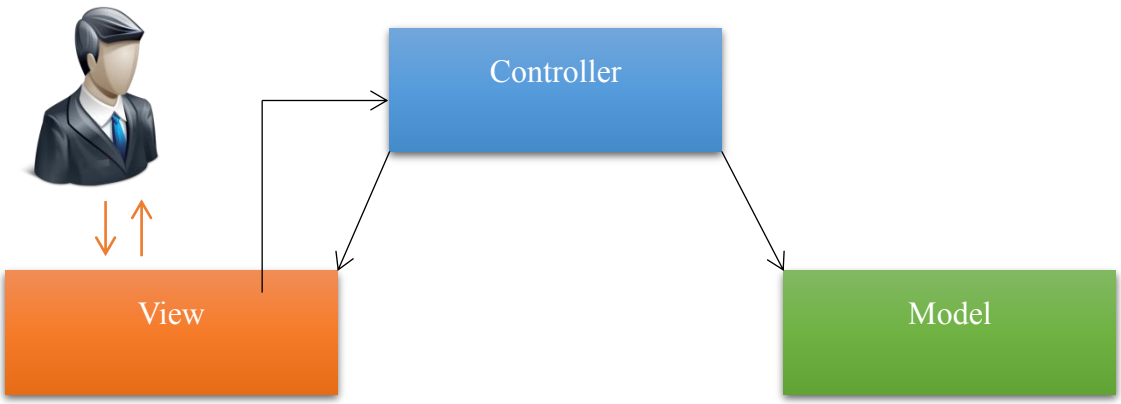
V podstatě každá implementace MVC je více či méně odlišná, každý autor ji pojímá jinak, avšak základní myšlenka je všem společná. Existuje několik variací MVC, které se liší hlavně v pojetí Controlleru, případně jeho vazby na View.

Myšlenka Model-View-Presenteru vychází z MVC, ale modifikuje úlohu Presenteru (Controlleru) a View. Presenter obsahuje nyní mimo aplikační i prezentační logiku (proto Presenter). View zpracovává vstup od uživatele přímo pomocí komponentů a volá při aktivitě konkrétní činnost Presenteru. [Bernard, 2009] Architektura je zachycena na Obrázek 2.

Jednou z implementací je dle [Fowler, a další, 2002] Passive View, který říká, že View je třeba udělat co nejvíce „hloupé“ a nesvěřovat mu více, než jen zobrazování dat. Presenter má teď daleko větší zodpovědnost, protože musí obstarat svoji původní funkci a navíc ještě nepřímou komunikaci mezi Modelem a View. Jednou z velkých výhod tohoto přístupu je podstatně lepší testovatelnost (není nutné testovat view, protože ten nemůže de facto nic zkazit). [Bernard, 2009] Architektura je zachycena na Obrázek 3.



Obrázek 2 – Diagram MVP. Zdroj: Autor dle [Bernard, 2009]



Obrázek 3 – Implementace MVP ve variaci Passive View. Zdroj: Autor dle [Bernard, 2009]

Pro ilustraci opodstatnění tohoto stručného výkladu, MVC architekturu implementuje framework Grails a MVP Passive View Nette. Toto vysvětlení pojmů je pouze velmi stručným shrnutím a není cílem zde poskytnout kompletní výklad[4], který by byl dalece nad rámec tohoto textu.

1.8.7 Platforma

Pro účel tohoto textu budu chápat termín platforma jako označení pro kombinaci jazyka a frameworku.

[4] Velmi kvalitní zdroj k problematice je [Fowler, a další, 2002], případně [Bernard, 2009]

2 Vybrané platformy pro vývoj

V této kapitole se budu věnovat velmi stručnému popisu jazyků, resp. platforem, které jsem zvolil pro porovnání. Jedná se o bezesporu dvě velmi časté volby vývojářů, analytiků a architektů při výběru platform pro vývoj webových aplikací. Pro srovnání jsem vybral jazyk PHP[5] s frameworkem Nette[6] a jazyk Groovy[7] s frameworkem Grails[8]. V následujících podkapitolách seznámím stručně čtenáře s těmito dvěma platformami.

2.1 Dynamické jazyky

Při výběru jazyků pro porovnání jsem ne zcela náhodně vybral jazyky dynamické (nikoliv dynamicky typované, to je jen PHP, Groovy může být i nemusí).

Klíčové vlastnosti dynamických jazyků definoval [Subramaniam, 2008] takto:

- Celkově méně boilerplate[9] kódu
- Většinou možnost modifikovat aplikaci za jejího běhu
- Většinou slabé dynamické typování (není třeba uvádět datový typ, ten bude zjištěn při běhu aplikace, samozřejmě na úkor výkonu)
- Větší míra abstrakce
- Rychlejší vývoj
- Větší produktivita práce
- Možná nebezpečí plynoucí z absence typových kontrol
- Větší potřeba testování (obzvlášť jednotkového)
- Kratší zdrojový kód

Výčet základních vlastností dynamických jazyků nelze samozřejmě aplikovat na každý takto orientovaný jazyk, prakticky každý má nějaké odlišnosti. Většina z těchto tvrzení bude však pravdivá.

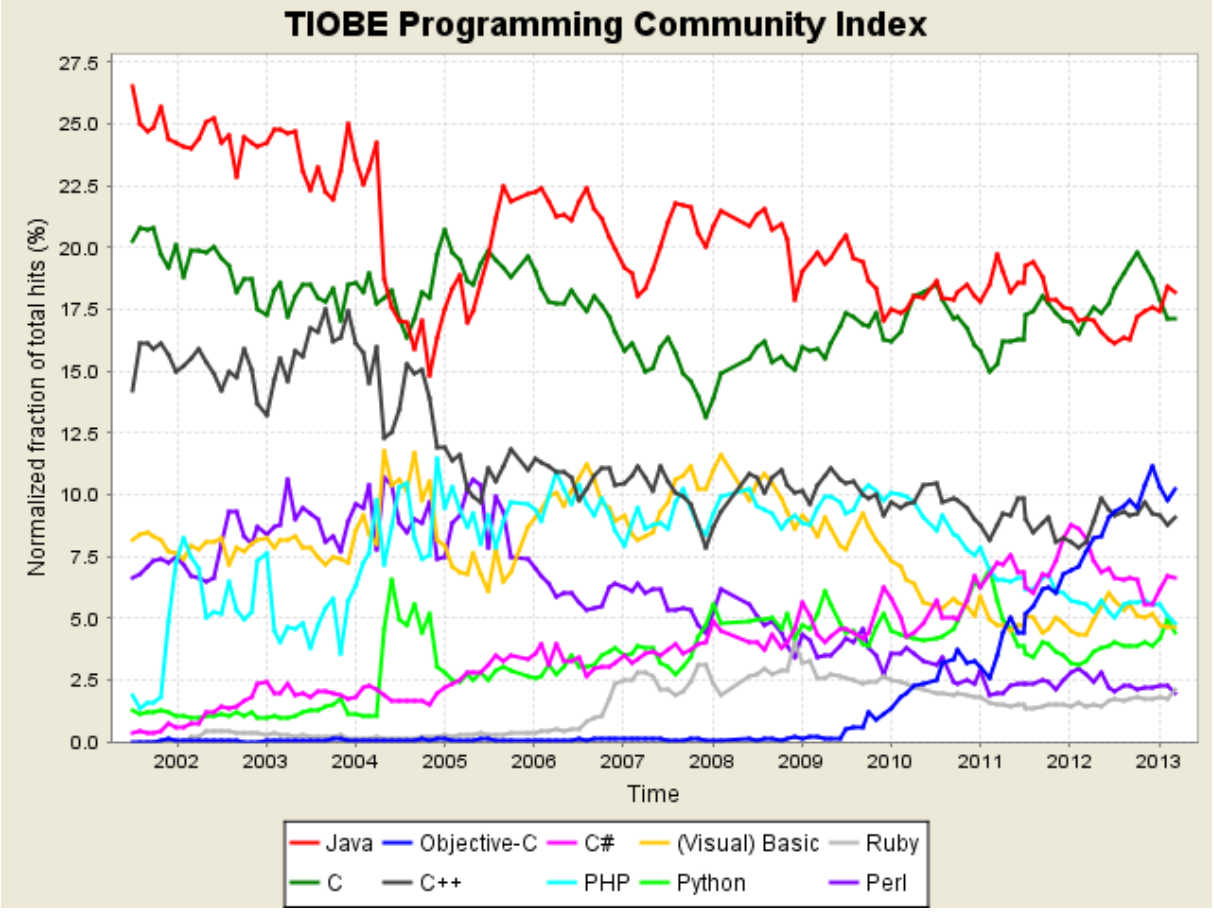
V dnešní době jsou dle [VersionOne, 2010] velice populární agilní metodiky vývoje (Scrum, XP, Lean…), které jsou více tolerantní ke změnám, vyžadují rychlý vývoj a zatahují

[5] Více informací o jazyce PHP je možné nalézt na oficiálních stránkách http://www.php.net
[6] Více informací o českém frameworku je možné nalézt na oficiálních stránkách http://www.nette.org
[7] Více informací o jazyce Groovy je možné nalézt na oficiálních stránkách http://groovy.codehaus.org
[8] Více informací o frameworku Grails je možné nalézt na oficiálních stránkách http://www.grails.org
[9] Programový kód, který je nutný k vykonání více či méně triviální funkcionality a navíc se poměrně často opakuje

zákazníka do procesu vývoje. Domnívám se, že i nástup agilních metodik podpořil rozvoj a zvýšil popularitu dynamických jazyků.

2.1.1 Popularita dynamických jazyků

Abych lépe ilustroval velké rozšíření dynamických jazyků, rozhodl jsem se využít statistik od společnosti Tiobe, které jsou založeny na tzv. Tiobe Indexu, jehož výpočet probíhá na základě sumy dotazů směřovaných na daný jazyk ve vyhledávačích dělených celkovým počtem dotazů na programovací jazyky.



Obrázek 4- Tiobe Index. Zdroj: [Tiobe Software, 2013]

Obrázek 4 zachycuje vývoj indexu [10]v letech 2002 až 2013. Z grafu je vidět, že v první desítce jazyků jsou 4 jazyky dynamické, což samo o sobě svědčí o velké popularitě. Jazyk Groovy se nachází na místě padesátém.

[10] Tato data byla zjištěna 25. listopadu 2012

2.2 Výběr frameworků

Podstatou tohoto textu není rozbor problematiky frameworků, tudíž zde uvedu, proč je důležité zabývat se výběrem frameworku a stručně svůj výběr zdůvodním.

Volba frameworku je stejně důležitá jako volba jazyka. Může vývojáři ušetřit mnoho práce, ale též znepříjemnit vývoj při špatném výběru. Domnívám se, že v dnešní době nemá smysl psát běžné aplikace, zvláště pak ty webové, bez využití frameworků, které nám podstatně ulehčují práci [Vrána, 2011], zrychlují vývoj a ve většině případů, pokud se jedná o dobrý framework, je finální kód podstatně kvalitnější (a také obsahuje best practices pro danou oblast) a s méně chybami. [Fowler, a další, 2002]

V případě Groovy je situace poměrně jednoduchá, existuje jeden velmi rubustní a kvalitní framework jménem Grails, který v sobě integruje některé funkcionality javovského frameworku Spring, entitně relačního mapperu Hibernate a dalších knihoven. První verze frameworku se objevily okolo roku 2006, takže jde o poměrně zralé a odladěné řešení. Pro Groovy jsem tedy vybral Framework Grails.

Volba u jazyka PHP je podstatně náročnější. Jazyk má větší uživatelskou základnu a tak pro něj existuje větší množství kvalitních frameworků. Zde záleží spíše na preferencích jednotlivých uživatelů. Obecně patří mezi velmi populární ve světě Zend Framework, Code Igniter, Yii, CakePHP a Symfony [PHPZag, 2012]. U nás je v posledních letech velmi populární Nette Framework od českého autora Davida Grudla. Troufám si tvrdit, že se jedná o velice inovativní framework implementující všeobecně uznávané návrhové vzory a přístupy pro vývoj webových aplikací [Grudl, 2012] (MVP, KISS, DRY, Dependency Injection a další). Tento Framework jsem zvolil proto, že porovnání s Groovy a Grails ještě nikdo neprovedl a domnívám se, že výsledky by mohly být velmi zajímavé.

2.3 Stručná historie a charakteristika PHP

Na úvod je třeba říci, že PHP není programovací jazyk, nýbrž skriptovací. Je to takzvaný interpretovaný jazyk překládaný za běhu interpreterem. Pokud bych se měl zabývat výkladem zkratky PHP, pak je to složenina Hypertext Preprocesor neboli „předzpracovávač“ hypertextových odkazů.

PHP existuje již od roku 1994, tehdy pod jménem PHP Tools (zkratka z Personal Homepage Tools), což byla jednoduchá sada CGI knihoven určená pro zjišťování

návštěvnosti osobních stránek autora jazyku, Rasmuse Lerdorfa. Původní verze byla napsaná v Perlu, později však upřednostnil C. Postupem času byly do knihoven přidávány další a další funkce, až vznikl podstatně významnější balík, který uměl pracovat například i s databázovým systémem. V roce 1995 autor uvolnil zdrojové kódy do komunity nadšených uživatelů, kteří opravovali chyby a podíleli se na jeho dalším vývoji. Tato verze bývá označována jako 1.0. [The PHP Group, 2012]

Ještě téhož roku se autor rozhodl změnit název na Forms Interpreter. Tato nová verze byla lehce podobná PHP, jak ho známe dnes. Syntaxe jazyka vycházela z Perlu a C. Později se inspirovala i objektovou Javou. I přes to, že se již jednalo o jazyk, odborná veřejnost ho za jazyk ještě nepovažovala, stále to byla „jen“ CGI knihovna (lépe řečeno modul) do webového serveru. [The PHP Group, 2012]

Brzy následovala velká změna. Zdrojové kódy autor kompletně přepsal a Forms Interpreter se přejmenoval zpět na PHP (Personal Home Page Construction kit). Syntaxe se začala podobat více majoritnímu jazyku doby, tedy C, aby se ho vývojáři mohli snadněji naučit. Do této doby interpreter běžel pouze pod Unixem či Posixem, začínalo se však uvažovat i o portaci na Windows. [The PHP Group, 2012]

Dalším milníkem byl rok 1996. Jméno se opět změnilo na PHP/FI, což je kombinace dvou předchozích názvů vydání (releasů). V tuto dobu pravděpodobně došlo k povýšení v očích veřejnosti, co se týče chápání samotného produktu, již nebyl brán jako pouhá sada knihoven, ale jako skutečný interpretovaný jazyk. Zcela zásadní byla nová podpora pro širokou škálu databázových systému (PostgreSQL, mSQL, DBM), podpora tvorby funkcí, cookies a mnoho dalších věcí, které zformovaly plnohodnotnou platformu pro vývoj zejména webových aplikací. Tato verze je považována za verzi 2.0. [The PHP Group, 2012]

V roce 1998 dva izraelští vývojáři Andi Gutmans a Zeev Suraski kompletně přepsali parser jazyk. Spolu s Rasmusem se pak rozhodli vytvořit nový jazyk, který bude rychlejší a mocnější než PHP/FI. Koncem roku vydali novou verzi, která se již jmenovala pouze PHP a nesla označení 3.0. Tato verze byla už velice podobná té dnešní. Přinesla podporu objektově orientovaného programování. Nutno však podotknout, že ne se všemi vlastnostmi[11], které by mít měla. Uživatelská základna se významně rozrostla a proběhla i portace na operační systém Windows. [The PHP Group, 2012]

[11] Příliš mnoho procedurálních a navíc nekonzistentně nazývaných funkcí. Absence referenčního odkazování na objekty a další problémy, které se řešily až v následujících verzích.

Izraelší vývojáři Andi a Zeev zakládají společnost Zend, která se v budoucnu stane velmi významnou díky většinovému podílu na vývoji PHP, jejich frameworku Zend a dalších produktů okolo PHP. V roce 1999 přepsali kompletně jádro PHP, které nazvali Zend Engine. Bylo podstatně modulárnější, rychlejší a dokonalejší než předchozí verze. Vznikl tedy nový release s označením PHP 4, který je občas k vidění i v dnešní době. [The PHP Group, 2012]

Nejzásadněji pak byla v poslední době vnímána verze PHP 5. Jádro bylo vylepšeno a vznikl Zend Engine 2, který opět zvýšil výkon a hlavně, přinesl nový objektový model. Opravoval mnoho chyb předchozí verze a opět přidal nové funkce. Ve verzi 5.3 se objevila i podpora pro jmenné prostory. PHP se stal jedním z nejpoužívanějších jazyků na světě. [The PHP Group, 2012]

2.3.1 Charakteristické rysy

PHP je objektově orientovaný dynamický jazyk se slabým typováním proměnných. To znamená, že datový typ je určen na základě jejího obsahu až při spuštění skriptu. Dynamičnost s sebou nese nejen výhody (jednoduchost, duck typing a další), ale i nevýhody (rychlost). Primárně je určen pro vývoj webových aplikací, ale je možné v něm napsané programy spouštět i přímo přes příkazovou řádku zavoláním interpreteru. Nejedná se o jazyk kompilovaný, nýbrž jazyk interpretovaný až za běhu programu. Někdy je chybně označován jako jazyk programovací, jedná se však o jazyk skriptovací. Jeho výstupem není nikdy program, ale skript. [Macintire, a další, 2011] Nespornou výhodou je otevřenost, způsob licencování a multiplatformovost (je portován na nejrůznější systémy jako je Solaris, Linux, Risc a další). [The PHP Group, 2012] Významnou výhodou je i velké množství dostupných hostingů (například na rozdíl od Ruby a Javy, kde je výběr poměrně omezený).

Co se týče samotného jazyka, je velice podobný Perlu, Javě a C, ze kterých bylo při návrhu syntaxe také čerpáno. [The PHP Group, 2012] Jedná se o velice jednoduchý jazyk, s velmi strmou křivkou učení. PHP obsahuje mnoho zabudovaných funkcí, přičemž tento přístup může mnohým programátorům připadat příliš procedurální.

PHP je ve světě velmi populární. Proto najdeme ohromné množství návodů, hotových zdrojových kódů konkrétních funkcionalit a mnoho knihoven, které jsou spojeny s velmi aktivní komunitou.

Další často zmiňovanou zajímavou součástí je široká škála extenzí, které rozšiřují možnosti jazyka. Zvláště pak v oblasti podpory databázových systémů je na tom PHP velmi

dobře, podporuje přímo práci s více než sedmnácti systémy jako je například Mongo DB. [The PHP Group, 2012]

Dnes k vývoji PHP přispívá celá řada programátorů nejen ze společnosti Zend, ale i z aktivní komunity. Mezi nimi nalezneme například i Čecha Jakuba Vránu.

Poslední verze jazyka je 5.4.9. K dnešnímu dni je PHP použito k programování na straně serveru ve více než 78% případech. Nejčastější používaná verze je 5.2, následovaná verzí 5.3. [W3Techs, 2012]. Podobné výsledky demonstroval i [Coyier, 2010] dle ankety, které se zúčastnilo více než 18 tisíc lidí.

2.3.2 Framework Nette

V roce 2004 vytvořil český autor David Grudl unikátní projekt, který si kladl za cíl zrychlit vývoj aplikací redukcí často se opakujících činností, jako jsou u většiny webových projektů například programování komunikace s databází, caching, routování a spousta dalších rutin. Tak vznikl framework Nette, který je údajně nejpoužívanější v ČR. (přesná data o používaných frameworcích v ČR prakticky neexistují). [Grudl, 2012]

V průběhu let se okolo PHP sešla velmi aktivní komunita, která se účastní i samotného vývoje frameworku. Existuje pro něj i celá řada více či méně kvalitních doplňků dostupných přímo z oficiálních webových stránek.

Z charakteristických rysů samotného frameworku bych vyzdvihl několik vlastností či funkcionalit [Grudl, 2012]:

- Návrhový vzor Model-View-Presenter
- Zabudovaná část knihovny pro práci s databází NotORM[12]
- Existence doplňku ORM Doctrine[13]
- Komponentový přístup
- Propracovaný systém routování
- Laděnka (debugovací nástroj)
- Propracovaná cache
- Slabší podpora pro testování
- Validátory a filtry vstupů
- Makra a helpery (funkce zjednodušující práci v šablonách)

[12] Jednoduchá knihovna pro práci s databází od českého autora Jakuba Vrány
[13] Doctrine je pravděpodobně nejlepší ORM vrstva pro práci s databází v PHP. Využívá ji i Symfony Framework.

- Dependency Injection kontajner
- Neon[14] konfigurace
- Velice dobrá podpora lokalizací
- Velice dobrá podpora pro AJAX i AJAJ

Aktuální verze frameworku je 2.0.10. Celý balík skriptů je vyvíjen primárně jedním člověkem, což je výhoda i nevýhoda. Pozitivním na tomto faktu je poměrně rychlý a přímočarý vývoj, negativním pak to, že existence a vývoj frameworku podléhá rozhodnutím jediného vývojáře, který si kdykoliv může říct, že už nemá zájem pokračovat a navíc se nemůže vývoji věnovat na plno. Naštěstí jsou alespoň zdrojové kódy volně přístupné i s poměrně benevolentní licencí.

Na oficiálních stránkách je k dispozici fórum v českém jazyce s velmi aktivní komunitou. Horší je to však v případě dokumentace, která rozhodně neposkytuje odpovědi na většinu otázek, které mohou programátora při vývoji napadnout (na rozdíl třeba od CakePHP nebo Zendu, kde je dokumentace skvěle zpracovaná a odpoví na velkou část otázek). Je proto nutné velmi často využívat fórum.

2.4 Stručná historie a charakteristika Groovy

Počátek vzniku dynamického jazyka Groovy, postaveného nad Javou, se datuje kolem roku 2003, kdy Bob McWhirter a Jason Strachan poprvé uveřejnili myšlenku vývoje nového jazyka na svém blogu a začali na něm postupně pracovat. Později v květnu 2004 byla specifikace jazyka dokonce uveřejněna do JCP (Java Community Process) jako JSR 241[15]. Naneštěstí byl vývoj poměrně náročný a téměř se od něj upustilo, nebýt Guillauma Laforge a Jeremy Raynera, kteří se připojili do procesu a rozhodli se překonat počáteční nezdary při vývoji. Nejprve se jim podařilo opravit základní chyby, které vyvíjený jazyk obsahoval a celkově ho stabilizovat do použitelné podoby. Vývoj byl ale poměrně komplikovaný a veřejnost už pomalu přestávala ve vznik tohoto jazyka věřit. K vývoji se opět přidalo několik velmi šikovných programátorů, kterým se spolu s Laforgem a Raynerem podařilo po dalších asi dvou letech[16] jazyk dodělat a vydat první verzi. [Subramaniam, 2008]

Jazyk postupně nabýval na významu, oblíbilo si ho mnoho vývojářů, na prezentacích a konferencích po celém světě byl velice úspěšný a odbornou veřejností byl kladně přijat.

[14] Neon je formát konfiguračních souborů od autora frameworku
[15] K dispozici zde http://www.jcp.org/en/jsr/detail?id=241#orig
[16] Konkrétně 2. ledna 2007

Postupně začala vznikat verze 1.1, která opravovala chyby z předchozí verze a byla relativně velkým skokem kupředu[17]. Poměrně rychle po vydání této verze byla přejmenována na verzi 1.5, především z důvodu velkých změn, jež nový název lépe reflektoval. [Wikipedia, 2012]

V listopadu 2008 kupuje společnost existující okolo jazyka Groovy velká a úspěšná firma Spring Source stojící za Spring frameworkem (později koupená společností VMWare). Tato akvizice posunula vývoj Groovy kupředu z několika důvodů, a to zejména možností čerpat finanční prostředky na vývoj a bohaté zkušenosti z vývoje Springu (využité zejména u vývoje frameworku Grails). [Subramaniam, 2008]

Následovaly verze 1.6, 1.7 a 1.8. V červenci 2012 byla vydána verze 2.0 a v plánu do budoucna jsou verze 2.1, 3.0 s 4.0. [Spring Source, 2012]

2.4.1 Charakteristický rysy

Groovy je dynamický jazyk postavený nad Javou a stejně jako ona je také považován za interpretovaný jazyk. Překlad jazyka Groovy probíhá do java bytecodu a poté je při spuštění interpretován JVM. Díky tomu jej lze spustit všude tam, kde Javu[18]. [Spring Source, 2012]

Syntakticky se při vývoji velkou mírou inspiroval u dalších dynamických jazyků, zejména Ruby, Scala a Pythonu. Zápis příkazů je tedy například oproti Javě, C a dalším, poměrně úsporný s velkou mírou tzv. syntactic sugar. [Spring Source, 2012]

Velkou výhodou je, že se nejedná o samostatný jazyk, ale je možné v něm používat jazyk Java, jeho knihovny a zápis kódu je možné kombinovat dle potřeby. Groovy je samozřejmě jazyk objektově orientovaný se všemi jeho aspekty. Zajímavostí je možnost využívat jak dynamického typování, tak explicitního statického. Samotný vývoj je poměrně přímočarý a křivku učení bych bez jakýchkoliv problémů srovnal s PHP (tedy velmi strmá) [Subramaniam, 2008].

Groovy je jazyk, ve kterém lze programovat webové i desktopové aplikace. Má také své GUI knihovny založené na Swingu. Nutno zmínit, že oproti Javě je Groovy podstatně pomalejší, zejména pokud využívá své přirozené dynamické typování proměnných. [Oliver, 2012]

---

[17] Podpora Javy 5, anotací, generických datových typů, výčtů a samozřejmě výkonové optimalizace a mnoho dalšího viz [Spring Source, 2012]

[18] Java Virtual Machine – virtuální stroj, který dokáže spouštět programy přeložené do java bytecode, což je kód nezávislý na platformě

Velmi vyspělá je zde podpora testování. Zde je naprosto jasně vidět, jak se autoři snaží tento proces vývojářům velmi usnadňovat, kde je to jen možné. Nejvíce patrné je to až v Grailsech, kde si troufnu tvrdit, že podpora testování je přímo famózní (ve srovnání například s PHP a Nette).

2.4.2 Framework Grails

Grails je velice pokročilý webový framework založený na jazyce Groovy. Už spojení těchto dvou technologií předurčuje tuto platformu pro agilní vývoj. Grails (původní název Groovy on Rails) se silně inspiruje u Railsů, což je webový framework pro jazyk Ruby, který je v agilním vývoji považován za průkopníka.

První verze frameworku byla vydána v roce 2005, další velice rychle následovala v roce 2006 a 2007 (za necelé dva roky čtyři verze). Avšak verze 1.0 až v únoru 2008.

Nutno podotknout, že vývoj je silně ovlivněn existujícími produkty firmy Spring Source a i firmou samotnou. Framework je vystavěn z velké části na základech Springu, využívá části MVC, Hibernate[19], WebFlow a další. Najdeme zde i produkty jiných vývojářů, webový kontajner Jetty či dekorátor SiteMash. Faktem je, že Grailsy tyto části pro své potřeby upravují. I přesto se jedná, dle mého názoru, o vynikající spojení, protože produkty od Springu jsou vyzkoušené, populární a hlavně zralé. Grails tedy netrpí příliš velkými neduhy, ale těží z best practices a zkušeností z vývoje u Springu a dalších zmíněných vyspělých produktů. [Judd, a další, 2008]

Pokud bych měl shrnout hlavní klíčové charakteristické rysy, budou to tyto [Judd, a další, 2008] [Hradecký, 2009]:

- Téměř žádná XML konfigurace díky myšlence Convention over configuration (COC)
- Pokročilý generátor kódu velmi urychlující práci
- Grails object relational mapping (GORM), což je ve své podstatě Hibernate
- Přímá podpora AJAXu
- Šikovný scaffolding pro podporu prototypování či testování
- Šablonovací systém GSP s vlastními tagy (obdoba JSTL)
- MVC architektura na základech Spring MVC[20]

[19] Objektově relační mapper [RedHat, 2012]
[20] Framework implementující návrhový vzor MVC bývá používán jako základ webových aplikací ve Springu. [SpringSource, 2012]

- Integrace Spring Web Flow[21]
- Podpora zápisu konfigurace anotacemi
- DI Framework Spring
- Velice dobrá podpora pro jednotkové i integrační testování
- Bohatá základna pluginů
- Zázemí velké společnosti
- Možnost placené podpory
- Velmi dobrá podpora lokalizací, validací a filtrů
- Podpora routování

[21] Framework pro management toku aplikace. Umožňuje elegantně řešit vícekrokové scénáře, pracovat se stavy aplikace a procesně řídit pohyb uživatele v aplikaci. [SpringSource, 2012]

3 Kritéria pro srovnání

V této kapitole se budu zabývat definicí a popisem ucelené sady kritérií, pomocí kterých by mělo být možné ohodnotit ve své podstatě jakýkoliv dnešní moderní framework pro rychlý vývoj aplikací. Tato sada kritérií by měla být dostatečně komplexní, aby zohlednila všechny, nebo přinejmenším alespoň základní aspekty takového frameworku.

Na základě analýzy několika existujících frameworků (CakePHP [Cake Software Foundation Inc., 2013], Grails [Spring Source, 2012], Nette [Grudl, 2012], Zend [Zend Technologies, 2013], Rails [Rails Core Team, 2013]), a dalších prací, ([Hradecký, 2009], [Odehnal, 2011], [Zorgdrager, 2010], [Hanyš, 2010], [Zilvar, 2010]) zabývajících se srovnáním v této oblasti, jsem identifikoval kritéria, která považuji za zcela zásadní.

V následujících podkapitolách jednotlivě každé z nich popíšu a provedu jeho zhodnocení z pohledu obou frameworků. Metoda hodnocení bude stylem „jako ve škole“, to znamená, že budu hodnotit číselně od 1 do 5. (1 – nejlepší, 5 - nejhorší). Při samotném udělování známek se budu snažit postupovat maximálně objektivně, ale subjektivní složku naneštěstí zcela vyloučit nelze.

3.1 Dostupnost

Každý framework vyvíjí jednotlivec, nebo firma a zpravidla ho publikuje. Z toho plyne, že se jedná o produkt, který může být buď volně dostupný, placený či může být model nějakým způsobem kombinovaný (volně dostupný s placenou podporou atd.). Zároveň je nutné se také zabývat licenční politikou, tj. tím, za jakých podmínek můžeme využívat/modifikovat daný framework a zda nejsou na aplikaci, pracující s tímto frameworkem, kladena nějaká omezení.

Lze říci, že dostupnost frameworku nebývá zpravidla problémem. Velká část je jich dostupná zdarma a je k nim možné přikoupit podporu, většinou formou ročního poplatku. Co se týče licenčních ujednání, tak se zde příliš často neobjevují žádné komplikace. Frameworky mívají poměrně volné licence, co se týče možnosti používání i případných modifikací. Příkladem licence mohou být New BSD, GNU GPL (Projekt GNU's Not Unix General Public License), Apache nebo MIT X11.

Zhodnocení

Oba frameworky jsou na tom v podstatě podobně. Nette je volně dostupný zdarma s poměrně volnou licencí New BDS nebo GNU GPL. Grails je také volně dostupný zdarma, uvolněný pod licencí Apache.

Tabulka 1 – Vyhodnocení kritéria dostupnost. Zdroj: Autor

Nette	Grails
1	1

3.2 Podpora

U některých frameworků je možné zakoupit podporu, což je určitá forma pomoci vývojářům, kteří mají problém a neví, jak jej vyřešit, či pouze chtějí zkonzultovat určitý dotaz s odborníkem. Výhodou těchto podpor je určitá garance odpovědi a zpravidla i její určité fundovanosti (oproti některým odpovědím k dotazům na fórech). Tyto formy podpor nebývají zdarma a je třeba za ně zaplatit poplatek, který bývá vybírán například měsíčně či ročně. Na dané období jsou pak garantovány určité možnosti podpory.

Zhodnocení

U Nette žádnou garantovanou podporu pořídit nelze. Ani si nedovedu představit, jak by David Grudl (autor) mimo vývoje frameworku, školení, odpovídání na fóru, svých projektů a dalších aktivit, ještě garantoval podporu. Tuto službu by pravděpodobně musel zastřešit někdo jiný, nebo ji zajistit nějakým jiným způsobem. Grails jsou na tom o poznání lépe. Stojí za ním velká firma, která poskytuje možnost tuto podporu zakoupit. SpringSource, potažmo VMWare nabízí několik druhů podpory lišících se rozsahem a samozřejmě cenou, která začíná na 700 a končí na 7000 dolarech ročně. Nutno podotknout, že služba není levná a pravděpodobně ji využijí spíše firmy, než jednotlivci.

Tabulka 2 – Vyhodnocení kritéria podpora. Zdroj: Autor

Nette	Grails
5	2

3.3 Architektura frameworku

Domnívám se, že každý framework pro rychlý vývoj by měl být vystavěn na základě nějakého architektonického vzoru. Tento přístup pak přináší celou řadu užitků (v případě správného výběru i implementace). V této práci se pohybujeme v prostředí webových frameworků, takže nejdůležitější budou vzory spadajících do kategorie Web Presentation. Pravděpodobně nejznámějším vzorem z této kategorie je vzor Model View Controller a jeho deriváty. Význam tohoto vzoru vyzdvihuje i [Fowler, a další, 2002]. Architektonických návrhových vzorů z kategorie Web Presentation je celá řáda. Troufám si však tvrdit, že MVC je z nich nejpoužívanější. Framework jako celek zpravidla implementuje celou řadu návrhových i architektonických vzorů. Toto kritérium si však klade za cíl ohodnotit základní vzor, který utváří prakticky celou podobu daného frameworku.

Framework by měl však daný návrhový vzor implementovat pouze v případě, že taková implementace přinese užitek a bude v souladu s cíli frameworku a problémy, které by měl takový vzor řešit.

Zhodnocení

Nette implementuje méně rozšířený návrhový vzor Model View Presenter, který je derivátem Model View Controlleru. Grails pak klasický, již zmiňovaný, Model View Controller. Frameworky jsou si tedy na první pohled velmi podobné a začínají se lišit až při práci s presenterem/controllerem a view, kdy každý z nich využívá jiný přístup. Nedomnívám se však, že implementace jednoho, či druhého vzoru přináší oproti konkurentovi významnou výhodu. Rozbor výhod a nevýhod tohoto přístupu by si však možná zasloužil bližší pozornost v samostatné práci.

Tabulka 3 – Vyhodnocení kritéria architektura frameworku. Zdroj: Autor

Nette	Grails
1	1

3.4 Práce s adresami

Každý webový framework je již z podstaty své existence nucen pracovat s adresami, z tohoto důvodu je nasnadě definovat a zhodnotit, jakým způsobem tyto aktivity provádí. Z adresy je nutné vyčíst zamýšlenou akci, kterou chce uživatel provést a identifikovat správně

parametry. Vzhledem k tomu, že dnešním trendem je využívání tzv. „cool url adres[22]“ (jinak také „pěkných adres“), framework by měl maximální možnou měrou tyto adresy podporovat. Vysvětlování významu tohoto typu adres je nad rámec této práce. Podotknu však, že jsou pouze adresy čitelnější a přehlednější pro uživatele.

Na základě [Microsoft, 2011], zdrojových materiálů k frameworkům a mých vlastních zkušeností jsem identifikoval dva základní přístupy:

1) Přepisování – url adresy jsou staticky přepisovány (často bez vědomí uživatele), místo adresy http://www.jirifrank.cz/clanky.html může být ve skutečnosti zobrazována adresa http://www.jirifrank.cz/index.php?action=clanky (tato pravidla bývají definována v souboru .htaccess nebo Web.config)

2) Mapování – z hlediska uživatele zde dochází ke stejnému jevu jako v předchozím případě. Rozdíl je pouze v tom, že adresy nejsou (pokud nepočítáme samotnou definici obecného mechanismus) přepisovány na základě pravidla vně aplikace, ale pouze mapovány na základě pravidel definovaných uvnitř aplikace. Tento přístup je možné členit na další dva typy.

    a. Jednosměrné – většinou dochází k mapování adresy na odpovídající akci na základě pravidla pouze při požadavku

    b. Obousměrné – zahrnuje předchozí přístup, tedy jednosměrné zpracování, ale navíc nabízí i reverzní, to znamená, že nejen z adresy na akci, ale i z akce na adresu (šikovné při generování odkazů atd.)

Nejvyspělejší je z těchto přístupů samozřejmě obousměrné mapování, které je nejen nejpohodlnější, ale i nejefektivnější přístup pro práci s adresami. Umožňuje centralizovaně řídit podobu adres, jejich lokalizaci, automatizovaně přidávat parametry a další užitečné výhody plynoucí z implementace.

Zhodnocení

Nette poskytuje poměrně vyspělý mechanismus pro práci s adresami podporující jednosměrné i obousměrné routování, omezení, skupiny, zástupné symboly, filtrování a další funkcionality. Z mého pohledu mu nelze nic vytknout a dá se říci, že jsem se dosud nesetkal s úkolem, který bych měl problém realizovat. V případě potřeby specifického chování nabízí Nette možnost napsat si vlastní třídu, či rozšířit stávající (SimpleRouter, Route).

[22] Tento pojem znamená využívání adres například v tomto formátu /controller/action/id namísto index.php?controller=a&action=b&id=c

Nejprve je nutné zmínit, že Grails oproti Nette nevyužívá termín Router/Routing, ale URL Mapping, tedy mapování. Principy obou termínů jsou však v tomto případě shodné. Možnosti Grails jsou prakticky shodné s Nette. V případě, že uživateli nevyhovuje výchozí mechanismus, je možné si napsat vlastní, podobně jako v Nette.

Tabulka 4 – Vyhodnocení kritéria práce s adresami. Zdroj: Autor

Nette	Grails
1	1

3.5 Convention over Configuration

Convetion over Configuration je princip, který česky řečeno říká, že je preferováno dodržování konvencí před explicitní konfigurací. V případě, že se snaží framework dodržovat tento přístup, mělo by být nutné dělat pouze velmi málo zásahů do konfiguračních souborů/direktiv pro konfiguraci rutinních úkolů. [Rocher, a další, 2009] Nelze však předpokládat, že vše bude nakonfigurováno samo. Z logiky věci, například databázové spojení je nutné vždy nastavit ručně. Tolik tedy jen pro upřesnění, co od tohoto přístupu očekávat či neočekávat.

Tím, že jsou definována pravidla pro pojmenování souborů typu controller, view nebo model lze velmi výrazným způsobem zmenšit počet nutných zásahů do konfiguračních souborů. Nemusí se jednat jen o pojmenování souborů, ale i tabulek v databázi nebo tříd.

Zhodnocení

Oba frameworky jdou v duchu této filosofie a je to znát na každém kroku. V případě dodržování jmenných konvencí, vše (myšleno veškeré standardní úlohy, které již ze své povahy nemusí nutně konfiguraci potřebovat) funguje prakticky bez jakéhokoliv zásahu do konfiguračních souborů.

Tabulka 5 – Vyhodnocení kritéria Convention over Configuration. Zdroj: Autor

Nette	Grails
1	1

3.6 Dependency Injection

Pro plné pochopení významu Dependency Injection je nutné nejprve vysvětlit princip Inversion of Control, pak samotné DI a také Dependency Injection Container.

Inversion of Control, neboli princip obrácené závislosti říká, že je občas výhodnější být volán, než v případě potřeby sám volat. Tento výrok se může zdát lehce zmatený, ale opak je pravdou. Pokud bych měl více přiblížit význam, jde ve své podstatě o to, jak předat odpovědnost za řízení někomu jinému, pro koho může být tato odpovědnost přirozenější. Příkladem mohou být metody setUp a tearDown u většiny testovacích frameworků. Při spuštění testů se vývojář nemusí starat explicitně o jejich volání, odpovědnost za jejich spouštění ve vhodnou chvíli je na frameworku (i když by bylo možné tyto metody volat explicitně při každém testu). Odpovědnost za volání těchto metod byla předána ze třídy samotné do třídy rodičovské, došlo k inverzi. Občas bývá také tento princip nazýván Hollywood principle se známou poučkou „Don´t call us, we will call you.“ (Nevolejte nám, my se vám ozveme). Stejně jako Martin Fowler i já se přikláním k tvrzení, že se jedná pouze o jiné pojmenování pro Inversion of Control. Význam tohoto principu tedy spočívá především ve volnějších vazbách mezi komponenty (loose coupling) a tím tedy i k lepší udržovatelnosti a testovatelnosti. [Fowler, 2005]

Jednou z implementací Inversion of Control (IoC) je Dependency Injection (DI), tzv. injektáž závislostí. Opět poněkud zjednodušeně řečeno říká: „Nestarejte se o to, kde vezmete potřebné závislosti, pouze je nadefinujte a někdo další vám je předá.“. Tato technika velmi dobře odstiňuje závislost na vytváření instance konkrétní třídy. Lze jednoduše definovat rozhraní a třídě pak předat danou třídu, která toto rozhraní implementuje. Opět jako u předchozího principu zde dochází k uvolnění vazeb mezi třídami. Jak je vidět, DI opravdu vychází z myšlenky IoC, předává odpovědnost za obstarávání potřebných závislostí někomu jinému. [Fowler, 2004] Další výhodou, kterou je nutnou zmínit v případě použití DI je zastavení vytváření skrytých závislostí, to znamená těch závislostí, které nejsou na první pohled vidět díky tomu, že jsou „zadrátované“ v útrobách dané třídy. [Grudl, 2012]

S Dependency Injection souvisí tak zvaný kontajner. To není zpravidla nic jiného, než třída, která se stará o tvorbu instancí tříd a jejich následnou injektáž, v podstatě automatizovaně. Kvalitní kontajnery umí vytvářet instance tříd v takovém pořadí, v jakém jsou potřeba. Většinou je nutné explicitně třídy, kde má docházet k injektáži, definovat včetně

dané metody (setter, constructor,…). Velmi často dochází k záměně mezi Dependency Injection a kontajnerem, což ale není to samé.

Pro zhodnocení, zda framework obsahuje „dostatečně dobrý DIC“, je třeba definovat několik kritérií. Jejich výběr jsem provedl dle [Pichlík, 2007]:

- Možnosti definování závislostí – jaké všechny způsoby definování aplikace podporuje (např.: xml, anotace, neon,…)
- Aplikovatelnost – na jakých místech v aplikaci lze injektáž využít
- Způsob nastavení závislostí – jaké metody je možné pro injektáž využít (např.: setter, property, constructor, interface)
- Autowiring – podpora automatizované injektáže bez explicitního vydefinování závislostí. Např.: lze provést anotaci proměnné nějakými informačními metadaty (např.: @inject, @autowire, type hint, apod.) a kontajner sám detekuje závislost a danou instanci třídy automatizovaně nainjektuje

Na závěr zdůrazňuji, že výše uvedená kritéria nemají za cíl postihnout či srovnat kompletní funkcionalitu kontajneru, ale pouze ty základní stěžejní možnosti, které by měla každá implementace DI podporovat. Toto téma je velmi široké a komplexní, naneštěstí zde však není prostor se mu plně věnovat ve všech jeho konsekvencích.

Zhodnocení

Grails v tomto kritériu dominuje na plné čáře. Spring kontajner, který je obsažen i v tomto frameworku, je velice kvalitní a dobře řeší všechna výše uvedená kritéria. Pravdou však je, že DI je při běžné práci ve frameworku využíváno podstatně méně (alespoň viditelně), než by mohlo být. Práce s kontajnerem je také přizpůsobena jazyku Groovy a jinému prostředí. Grails dokonce zavádí nový pojem Dependency Injection by Convention, což je ve své podstatě označení pro inteligentnější autowiring, který zafunguje nejen pro property injection, ale i pro setter injection. Závislosti lze jinak získávat i přes aplikační kontext.

Nette obsahuje také Dependency Injection Container, který však obsahuje podstatně méně možností než Spring a celkově ještě není implementace Dependency Injection dostatečně vyzrálá. Nutno podotknout, že kontajner je stále ve vývoji a v nově připravovaných verzích jsou v plánu opravy nedostatků typu nemožnost injektovat parametry, či injektáž na základě rozhraní a další. Závislosti je možné také získávat přes aplikační

kontext (návrhový vzor Service Locator), což ale není příliš doporučovaný přístup. Rozdíly nejlépe ilustruji na srovnávací tabulce.

Tabulka 6 – Vyhodnocení podkritérií kritéria Dependency Injection. Zdroj: Autor

Kritérium	Nette	Grails
Možnosti definování závislostí	config.neon	resources.groovy, resources.xml, anotace
Aplikovatelnost	služby, presentery	ve všech třídách
Způsoby nastavení závislostí	constructor, setter	setter, property (datový typ, ref konstrukt), constructor
Autowiring	Ano	Ano

Z tabulky rozdíl mezi oběma kontajnery možná nevypadá příliš markantně, ale je důležité nenechat se zmást, možnosti Springu, potažmo Grails, jsou na podstatně vyšší úrovni.

Tabulka 7 – Vyhodnocení kritéria Dependency Injection. Zdroj: Autor

Nette	Grails
3	1

3.7 Podpora testování

S rostoucí komplexitou aplikací roste i potřeba je testovat a to jak z důvodu ověřování, že fungují, jak bylo skutečně zamýšleno, tak i z důvodu kontroly samotné funkčnosti nebo ověřování funkčnosti po aplikovaných změnách.

U dynamicky typovaných jazyků roste potřeba testování dvojnásobně a to z toho důvodu, že kompilátor nebo interpreter odhalí podstatně méně chyb (díky dynamické povaze jazyka) než u statických. [Rocher, a další, 2009]

Testovat webové aplikace lze nejčastěji buď jednotkově (testujeme relativně oddělené nezávislé celky) nebo integračně (testování funkčnosti více celků dohromady). Framework

nebo jazyk pro testování nabízejí většinou testovací knihovnu, která psaní a spouštění testů velmi usnadňuje.

Kritériem tedy budou (uvažuji srovnání presence možnosti testování a podpory základních technik, nikoliv podrobný rozbor kvalit dané knihovny):

- Podpora jednotkového a integračního testování
- Testování všech částí frameworku
- Podpora pokročilých metod (fixture, mock, stub…)

Zhodnocení

Na základě výše definovaných kritérií jsem vypracoval srovnávací tabulku. Testovací knihovna pro Nette, tzv. Tester, je poměrně nová a stále ve fázi beta verze. V porovnání s PHPUnit (nejznámější knihovna pro testování aplikací napsaných v PHP) jsou možnosti značně omezené, ale je nutné vyzdvihnout určitou jednoduchost, s jakou se testy vytvářejí. Oproti tomu Grails nabízí rubustní testovací prostředí, které dokáže otestovat prakticky každou část aplikace a to i za pomocí velmi pokročilých metod, jejichž použití maximálně zjednodušuje anotováním.

Tabulka 8 – Vyhodnocení podkritérií podpory testování. Zdroj: Autor

Kritérium	Nette	Grails
Podpora jednotkového a integračního testování	Ano s určitými omezeními	Ano
Testování všech částí frameworku	Ne	Ano
Podpora pokročilých metod	Ne	Ano

Hodnocení je v tomto případě tedy jednoduché. Grails nabízí jednu z nejlepších podpor pro testování webových aplikací, s jakou jsem se zatím setkal. Nette je s testováním, resp. se svojí knihovnou pro testování, ještě v začátcích.

Tabulka 9 – Vyhodnocení kritéria podpora testování. Zdroj: Autor

Nette	Grails
3	1

3.8 Integrované funkce v základu

Každý framework obsahuje nějakou předpřipravenou množinu funkcí usnadňující práci s často se opakujícími úlohami. Pro příklad lze zmínit práci s řetězci, formátování, mailování a další. Tyto funkcionality mohou být samozřejmě řešeny v samotném jazyku. Jejich kouzlo však spočívá v tom, že dokáží zredukovat počet řádků kódu či zjednodušit přístup k dané funkcionalitě.

Na základě analýzy běžných funkcionalit několika frameworků (Zend, CodeIgniter, Nette, Grails a CakePHP) jsem vypracoval jednoduchou sumarizaci klíčových vlastností. Tyto funkcionality jsou natolik základní, že by dle mého názoru měly být standardní součástí každého frameworku. Níže tedy uvádím jmenný seznam těchto základních funkcionalit:

- Pokročilá podpora mailování
- Implementovaná cache
- Pokročilé zpracování obrázků (ořezávání, zmenšování, zvětšování, komprimace…)
- Podpora pro stránkování
- Podpora pro autorizaci a autentifikaci
- Funkcionalita logování
- Validace prvků
- Bezpečnostní mechanismy (ochrana před CSRF, XSS, SQL Injection)

Zhodnocení

Níže uvádím tabulku, která shrnuje stav jednotlivých kritérií v obou frameworcích.

Tabulka 10 – Vyhodnocení podkritérií kritéria integrované funkce v základu. Zdroj: Autor

Kritérium	Nette	Grails
Pokročilá podpora mailování	Ano	Ne (pouze základní)
Implementovaná cache	Ano	Ano (výchozí plugin)
Pokročilé zpracování obrázků	Ano	Ne
Podpora pro stránkování	Ano (bez prezentace, pouze logika)	Ano
Podpora pro autorizaci a autentifikaci	Ano (základní)	Ano (základní)

Kritérium	Nette	Grails
Funkcionalita logování	Ano	Ano
Validace prvků	Ano	Ano
Bezpečnostní mechanismy	Ano	Ano

Z tabulky je patrné, že Nette má o něco bohatší základní funkce. Grails tento problém však řeší velice snadno pluginy. Dále má Grails velice dobrou podporu pro práci s rozšířeními. Celkově tedy, dle tabulky, vychází v tomto bodě Nette o něco lépe.

Tabulka 11 – Vyhodnocení kritéria integrované funkce v základu. Zdroj: Autor

Nette	Grails
1	2

3.9 Dostupné běhové prostředí

Každá webová aplikace musí být spouštěna v nějakém prostředí, zpravidla se mu říká webový server, či webový kontajner v případě Javy EE/Grails. Zde je velký rozdíl mezi aplikacemi v PHP a Groovy/Grails. Aplikace v PHP je při každém požadavku klienta spuštěna znovu a následně je vygenerována skriptem (aplikací) odpověď, která je v nějaké formě vrácena klientovi. Webový server v tomto případě slouží spíše jako prostředník mezi uživatelem (resp. jeho požadavkem) a PHP interpreterem (resp. aplikací). Oproti tomu Grails aplikace jsou spuštěny v kontajneru permanentně a obsluhují požadavky, aniž by celá aplikace musela být po každém požadavku spuštěna znovu.

Dalším faktem je, že PHP aplikace jsou vázány na webové servery, které umí pracovat s PHP interpretrem. Naproti tomu Grails aplikace lze spustit prakticky ve všech webových kontajnerech, které jsou kompatibilní s Javou EE 5 (Servlet ve verzi 2.5 a vyšší). [Spring Source, 2012].

Webové servery, nebo kontajnery jsou odlišné od velmi komplexních a rubustních, až po jednoduché s minimálními nároky, ale i možnostmi. Domnívám se, že právě z tohoto důvodu je i tento fakt podstatný. U webových serverů a kontajnerů je zajímavé i to, zda se jedná o placený produkt, nebo je zdarma.

Zhodnocení

Paleta běhových prostředí je rozdílná u obou platforem. V současné době neexistuje jediná, která by podporovala běh obou dvou. Níže uvádím tabulky, které shrnují většinu nejznámějších webových serverů či kontajnerů pro dané platformy.

Tabulka 12 – Přehled dostupných běhových prostředí pro Grails. Zdroj: Autor a [Spring Source, 2012]

Název	Verze	Výrobce	Licence
Tomcat	6,7	The Apache Software Foundation	Open source
TC Server	-	VMWare	Komerční
Virgo		The Eclipse Foundation	Open source
Glassfish	2,3	Oracle	Open source/Komerční
Resin	3,4	Caucho Technology	Open source
JBoss	5,6,7	RedHat	Open source
Jetty	6,7	Webtide	Open source
WebSphere	6.1,7	IBM	Komerční
WebLogic	10,10.3,11,12	Oracle	Zdarma/Komerční

Tabulka 13 – Přehled dostupných běhových prostředí pro Nette. Zdroj: Autor

Název	Verze	Výrobce	Licence
Apache	2.0, 2.2 , 2.4	The Apache Software Foundation	Open source
Nginx	1	Igor Sysoev a komunita	Open Source
IIS	5, 6, 7	Microsoft	Komerční

Jak je vidět, možnosti v případě Grails jsou podstatně širší, a to jak komerčního charakteru, tak open source. Nutno podotknout, že možností v případě PHP je sice méně, ale jedná se o velmi kvalitní a vyzkoušené produkty. Apache je velmi kvalitní webový server, který běží na nejednom hostingu.

Tabulka 14 – Vyhodnocení kritéria dostupné běhové prostředí. Zdroj: Autor

Nette	Grails
2	1

3.10 AJAX/AJAJ

V posledních letech došlo k proměně webu ze statických stránek na stránky dynamicky reagující. Webové aplikace se mnohdy velmi podobají klasickým desktopovým. To je mimo jiné možné i díky technologií nazývané AJAX (Asynchronous Javascript and XML) nebo AJAJ (Asynchronous Javascript and JSON), což jsou v podstatě dva identické přístupy využívající v jádru stejnou techniku, kterou je tzv. XmlHttpRequest (XHR). XHR je funkce v javascriptu pro asynchronní přenos dat bez nutnosti znovu načíst stránku, nutno však podotknout, že je poměrně stará. Formáty přenášených dat tedy mohou být nejen nejpoužívanější formáty XML a JSON, ale lze přenášet i text či HTML. [Macintire, a další, 2011]

Díky těmto změnám, začaly být kladeny vyšší nároky i na frameworky, aby tento směr, kudy se webové stránky ubírají, začaly lépe podporovat. Cílem tohoto kritéria není porovnat přístupy jednotlivých frameworků, ale spíš dokumentovat existenci samotné podpory AJAXu a stručně tento fakt okomentovat a subjektivně ohodnotit.

Zhodnocení

Na tomto kritériu je názorně vidět, že každý framework se rozhodl ubírat naprosto odlišným směrem. Nette zavedlo koncept snippetů, které se identifikují klíčovým slovem snippet přímo v šabloně. Tento fragment může být pak jednoduše invalidován. Součástí tohoto konceptu jsou i signály, což ve své podstatě není nic jiného, než klasický request, který se však chová od klasického lehce odlišně. V případě běžného plnohodnotného requestu dojde při obnovení stránky k akci v aktuálním presenteru, k akci v jiném než aktuálním presenteru, znovu načtení celé stránky a dalším. V případě signálu může být obnovena celá stránka nebo určité snippety, a to vše pouze v rámci aktuálního presenteru. Jde tedy v podstatě o takový „malý“ request, při kterém zůstává aktivní daná akce presenteru, ale metodou handle se zpracuje daný signál, který mírně změní stav daného presenteru. Pro ilustraci lze uvažovat na příklad tabulku s výpisem úkolů. V případě kliknutí na tlačítko smazat se může provést signál, který zobrazí dialogové okno vyžadující potvrzení akce s tím, že zbytek stránky zůstává

v původním stavu. Výhodou je, že Nette signál provede i v případě, že nebude přítomen Javascript (AJAX). V případě, že vznikne potřeba nějaký odkaz provést ajaxovým způsobem, stačí mu dát pouze třídu ajax. Nette využívá ve výchozím stavu pro používání AJAXu knihovny jQuery [Grudl, 2012].

Domnívám se, že přístup, který zvolilo Nette je poměrně inovativní a velmi jednoduchý jak z hlediska použití, tak i pro pochopení a přitom velmi logicky zapadá do architektury aplikace v tomto frameworku.

Grails jakožto moderní framework má podporu AJAXu samozřejmě také, dokonce umožňuje používat nejen jQuery, ale i Dojo, Prototype, Google Web Toolkit, YUI a MooTools. Princip celé práce je zde odlišný. Namísto definice fragmentů a signálů se zde pomocí předpřipravených tagů a jejich atributů specifikují jednotlivé funkcionality, které jsou posléze mapovány na konkrétní obslužné rutiny daného Javascript frameworku. Jednoduše lze tedy vytvořit například odkaz provedený ajaxem se zobrazením vrácené hodnoty v nějakém boxu. To je ve své podstatě celá podpora, kterou Grails nabízí. Na první pohled může vypadat skromně, ale umožňuje jiným způsobem téměř to samé, co Nette.

Hodnocení dle mého názoru hovoří mírně ve prospěch Nette, jehož přístup se mi zdá příjemnější a zajímavější.

Tabulka 15 – Vyhodnocení kritéria AJAX/AJAJ. Zdroj: Autor

Nette	Grails
1	2

3.11 Persistentní a doménová vrstva

V rámci vrstvené architektury můžeme dle různých autorů (například dle [Fowler, a další, 2002]) členit aplikaci na tyto části:

- Persistentní – komunikace s ostatními systémy (často databázové systémy)
- Doménovou – obsahuje business logiku aplikace
- Prezentační – komunikace s uživatelem

Prezentační vrstva se často skládá ještě z části samotné aplikační logiky (často nazývána controller) a zobrazovací části (často nazývána view). Cílem této části je porovnat persistentní a doménové vrstvy.

U persistentní vrstvy zde není příliš velký prostor pro srovnání implementací pro dané databázové systémy. Proto jsem zvolil jako hlavní kritérium počet podporovaných systémů oběma frameworky.

Doménová vrstva si zaslouží hlubší porovnání. Zde je prakticky nejzásadnější, jakým způsobem daný framework k této problematice přistupuje. Konkrétně, zda je cílem poskytnout komplexní a zpravidla i komplikovanou vrstvu doménové logiky, či spíše jednoduchý, snadno použitelný nástroj. To je základní předpoklad pro správné ohodnocení tohoto kritéria. Detaily jednotlivých přístupů jsem popsal dle [Fowler, a další, 2002] a [Pecinovský, 2012]

JEDNODUCHÝ PŘÍSTUP

Tento směr řešení bude realizován nejčastěji pomocí velmi jednoduchého přístupu k persistentní vrstvě možností jednoduše pokládat dotazy, nebo sofistikovanějšími metodami, které popisují níže definované návrhové vzory:

- Transaction script – data jsou získávána typicky nějakou procedurou z jednoho místa, kdy prostředníkem bývá brána (některá ze specializací návrhového vzoru Gateway, tedy Data Row nebo Data Table), která obsahuje logiku pro získání dat a tato data pouze předává servisní třídě.
- Table Module – strukturovaný přístup k datům (na rozdíl od Transaction script), kdy jsou data organizována kolem tabulek (nikoliv procedur). Pracuje se zde často s tzv. RecordSetem, což je objekt simulující tabulku.

KOMPLEXNÍ ŘEŠENÍ

Komplexnější řešení se sice dají použít ve stejných situacích jako jednoduchá, ale dá se říci, že čím složitější projekt, tím vhodnější je adopce Domain Modelu a naopak, čím jednodušší, tím je výhodnější použít spíše Transaction Script nebo Table Module. Domain Model je pravděpodobně nejsofistikovanější řešení, co se týče zobrazování skutečné reality (řešené domény), které zpravidla podporuje objektově orientované aspekty návrhu aplikace.

Logika je zde strukturovaná, každý model (ve formě objektu) zodpovídá pouze za svoji konkrétní část. Daný objekt z reality zde tedy zastupuje právě jedna instance daného modelu.

Domain Model využívá pro práci s datovými zdroji v podstatě dvě základní možnosti definované návrhovými vzory Active Record a Data Mapper. Data Mapper se stará o mapování třídy/modelu na konkrétní tabulky. Active Record je v podstatě objekt, který reprezentuje jediný řádek tabulky a zapouzdřuje v sobě určitou logiku a základní operace s tímto řádkem. [Fowler, a další, 2002]

Dle mého názoru by bylo chybou hodnotit, který z těchto přístupů je obecně lepší. Každý z nich najde své uplatnění u konkrétního projektu. Samotný výběr pak bude záviset zpravidla na rozsáhlosti aplikace.

Výše popsané vlastnosti doménové vrstvy bude tedy dobré chápat spíše jako informativní vhled do problematiky. Samotnými kritérii bude poskytovaná funkcionalita, což bývají zpravidla ty vlastnosti, které jsou při výběru rozhodující. Tato kritéria jsem vybral na základě analýzy vlastností doménových vrstev několika existujících frameworků (Zend, CodeIgniter, Nette, Grails a CakePHP).

- Validace
- Customizovatelné chybové validační hlášky
- Události
- Vztah 1:1
- Vztah 1:M
- Vztah M:N
- Transakční zpracování
- Podpora dědičnosti
- Sofistikovaný přístup pro vyhledávání v datech – například fluent způsobem, kriteriálním přístupem či speciálním dotazovacím jazykem
- Přístup k návrhu Code First
- Přístup k návrhu Database First
- Podpora odlišných prostředí (produkční, vývojové atd.)

- Scaffolding – automatizované virtuální vygenerovaní celé aplikace

Zhodnocení

Jak jsem již nastínil, při popisu tohoto kritéria je důležité identifikovat cíl daného frameworku v oblasti doménové vrsty. Grails se snaží poskytnout skrze Hibernate velice komplexní nástroj (Domain Model, Data Mapper). Nette nechává v základu pouze Nette/Database, což je jednoduchý nástroj pro přístup k datům (Transaction Script) s tím, že lze samozřejmě doplnit například framework Doctrine 2 (Domain Model, Data Mapper), v základu však není.

Přehled oficiálně podporovaných databázových systémů uvádím v Tabulka 16.

Tabulka 16 – Podpora databázových systému oběma frameworky. Zdroj: Autor

Databázový systém	Nette	Grails
MySQL	Ano	Ano
Oracle Database	Ano	Ano
MS SQL Server	Ano	Ano
Postgre SQL	Ano	Ano
Firebird	Ne	Ano
DB2	Ne	Ano
Informix	Ne	Ano

Následující tabulka naopak shrnuje všechna podkritéria pro jednotlivé frameworky.

Tabulka 17 – Vyhodnocení podkritérií doménové vrstvy. Zdroj: Autor

Název	Nette	Grails
Validace	Ne	Ano
Customizovatelné chybové hlášky	Ano	Ano
Události	Ne	Ano
Vztah 1:1	Ano	Ano

Název	Nette	Grails
Vztah 1:N	Ano	Ano
Vztah N:M	Ne	Ano
Transakční zpracování	Ano	Ano
Podpora dědičnosti	Ne	Ano
Sofistikovaný přístup pro vyhledávání v datech	Ano	Ano
Přístup k návrhu Code First	Ne	Ano
Přístup k návrhu Database First	Ne	Ano
Podpora odlišných prostředí	Ano	Ano
Scaffolding	Ne	Ano

Celkové vyhodnocení tohoto kritéria je relativně dle očekávání ve prospěch Grails. Nette neposkytuje vyčerpávající podporu databázových systémů tak jako Grails, a ani neposkytuje některé základní prvky pro práci s databázovou vrstvou jako je podpora vztahu M:N a další. Oproti tomu se drží svého cíle (důležitost tohoto aspektu jsem zdůrazňoval již na začátku kapitoly), to znamená poskytnout jednoduchou a dobře použitelnou vrstvu, což se daří a nelze tedy hodnotit Nette příliš záporně kvůli absenci podpory přístupu Code First apod.

Tabulka 18 – Vyhodnocení kritéria persistentní a doménová vrstva. Zdroj: Autor

Nette	Grails
2	1

3.12 Šablonování

V případě, že aplikace pracuje s nějakými výstupy, je velice vhodné využít tzv. šablonování. Využití tohoto přístupu má přednosti v oddělení zobrazovací od aplikační logiky, bývá lépe spravovatelné, omezuje duplikace a zároveň jde v duchu s myšlenkou Separation of Concerns (princip oddělení zájmů) [Parr, 2004]. V zásadě bývá při implementaci využit některý z návrhových vzorů, které se zabývají touto problematikou,

například Template View (dle [Fowler, a další, 2002]), který je pravděpodobně jedním z nejvíce rozšířených.

Kvalitní šablonovací systém má zpravidla buď přímo či nepřímo zabudované další nástroje pro redukci často se opakujících úkolů nebo nástroje, které zpříjemňují práci. Takovými nástroji bývají zpravidla tzv. helpery (jednoduché funkce, které při zavolání mají za úkol vygenerovat kód nebo jednoduchou formou zpřístupnit funkci, která by měla jinak poměrně dlouhý zápis), které bývají implementovány různým způsobem. Buď jako samostatné příkazy ({form}) nebo formou html tagu (<t:form/>).

Další vlastností těchto nástrojů je rozdělení mechanismu šablony na layout (obecnou šablonou) a view (konkrétní pohled). Tento princip omezí duplikaci kódu na nezbytné minimum.

Na základě rozboru, který provedl Fabien Potencier [Potencier, 2009], autor PHP frameworku Symfony, a pak dále na základě analýzy několika dalších frameworků odkazovaných již v předešlých částech práce (Zend, CodeIgniter, Nette, Grails a CakePHP), jsem definoval základní sadu kritérií pro ohodnocení šablonovacího systému:

- Předávání proměnných
- Podpora layoutu a jednotlivých view
- Vkládání obsahu z view do různých míst v layoutu (bloky)
- Helpery
- Vkládání pomocí speciálních tagů nebo příkazů
- Kratší kód oproti manuálnímu výpisu
- Automatické escapování proměnných

Zhodnocení

Oba dva frameworky disponují vyspělým šablonovacím systémem. Grails staví na dekoračním enginu SiteMesh, Nette používá své vlastní řešení. Pro účely tohoto srovnání budu uvažovat makra u Nette jako součást helperů. Dále uvádím tabulku, která shrnuje stav jednotlivých kritérií v obou frameworcích.

Tabulka 19 – Vyhodnocení podkritérií pro zhodnocení kritéria šablonování. Zdroj: Autor

Název	Nette	Grails
Předávání proměnných	Ano	Ano

Název	Nette	Grails
Podpora layoutu a jednotlivých view	Ano	Ano
Vkládání obsahu z view do různých míst v layoutu (bloky)	Ano (libovolně)	Ano (omezeně)
Helpery	Ano (velmi mnoho)	Ano (základní funkce)
Vkládání pomocí speciálních tagů nebo příkazů	Ano (příkazy i tagy)	Ano (tagy)
Kratší kód oproti manuálnímu výpisu	Ano (o hodně)	Ano
Automatické escapování proměnných	Ano	Ano

Z tabulky je patrné, že Nette má velice vyspělý šablonovací systém, který si prakticky v ničem nezadá s Grails a dokonce ho v některých aspektech překonává. Domnívám se, že šablonovací systém Latté je u Nette jednou z jeho největších domén. Na druhou stranu lze říci, že i když v tomto Nette vede. Grails má šablonování také na velmi dobré úrovni a splňuje, dle mého názoru, všechny nároky, které jsou kladeny na dobrý šablonovací systém.

Tabulka 20 – Vyhodnocení kritéria šablonování. Zdroj: Autor

Nette	Grails
1	2

3.13 Možnosti rozšíření

Nedílnou součástí dobrého frameworku je i možnost rozšířit ho o již hotová řešení ve formě pluginů/modulů/komponentů, které jsou dílem komunity nebo autorů samotných. Oba frameworky podporují rozšíření. Toto kritérium by tedy mělo hodnotit níže definované body:

- Jednoduchost instalace doplňků
- Oblasti, pro které je možné doplňky instalovat (komponenty, pluginy, makra…)
- Existence portálu s doplňky
- Počet doplňků
- Kvalita doplňků

Zhodnocení

Oba dva frameworky umožňují modifikace jakékoliv své části a lze tedy vyměnit či upravit v podstatě cokoliv. Tento úhel pohledu však není podstatou tohoto kritéria. Cílem je zhodnotit standardní rozšíření, která se dají jednoduše stáhnout a nějakým způsobem vložit do aplikace. Tabulka s přehledem jednotlivých dílčích kritérií a jejich vyhodnocením je níže. Při hodnocení budu uvažovat pouze oficiální zdroj doplňků.

Tabulka 21 – Zhodnocení jednotlivých podkritérií pro možnost rozšíření. Zdroj: Autor

Název	Nette	Grails
Jednoduchost instalace doplňků	Velmi dobrá (automatizace instalace pomocí Composeru pouze někde, jinak je potřeba manuálně doplněk stáhnout a vložit např. do adresáře libs)	Výborná (možnost instalovat vše skrze závislosti)
Oblasti, pro které je možné doplňky instalovat (komponenty, pluginy, makra…)	Komponenty, helpery, různé další nástroje a rozšíření (přihlašování pomocí Facebooku, lokalizace…)	V podstatě vše
Portál s doplňky	Dobrý (v plánu je nová verze)	Výborný
Počet doplňků	Průměrné množství (oficiálně okolo 100 doplňků)	Velké množství (oficiálně více než 350)
Kvalita doplňků	Dobrá (u některých doplňků velký výskyt bugů, nefunkčnost v nových verzích frameworku)	Vysoká úroveň (proces kontroly kvality, celá řada doplňků od SpringSource a renomovaných přispěvovatelů)

Z tabulky výše je patrné, že Grails si vede, co se týče doplňků, podstatně lépe. Pluginů je velké množství a jsou poměrně kvalitní. Zázemí v podobě portálů je také na výborné úrovni. Pluginy též existují prakticky pro jakoukoliv část frameworku. V tomto kritériu jsem nezohledňoval existenci doplňků v Javě, které jdou v Groovy, resp. Grails zpravidla bez problémů použít. V případě zohlednění tohoto aspektu by Grails získal ještě podstatně širší

možnosti rozšíření. Nette je v oblasti doplňků poměrně na začátku. Standardy, které by upravovali náležitosti pro pluginy prakticky neexistují, automatizace v podobě správy závislostí nástrojem Composer (přes repositář Packigist) byla zavedena relativně nedávno a velkou část doplňků tedy nelze tímto stylem instalovat. Portál v aktuální podobě je sice ucházející, ale zasloužil by si vylepšení (beta verze nového portálu existuje).

Tabulka 22 – Vyhodnocení kritéria možnosti rozšíření. Zdroj: Autor

Nette	Grails
3	1

3.14 Podpora lokalizací

V dnešním velmi globalizovaném světě často vzniká potřeba aplikaci lokalizovat do jednoho či více dalších jazyků. Možnost řešit tento fakt n-násobnou replikací stránek lze vyloučit jako značně neefektivní a v případě modifikací vzhledu, funkcí a dalších věcí jako velmi neefektivní. Frameworky proto v řadě případů přicházejí s možnostmi pro řešení tohoto úkolu. Dobře zpracovanou podporu lokalizací považuji osobně za jednu z klíčových vlastností kvalitního frameworku.

Framework by měl být schopen podpořit statickou lokalizaci v šablonách, informačních a validačních hlášeních a dále pak ideálně nabídnout mechanismus pro řešení lokalizací modelu. Základními kritérii tedy jsou:

- Statické lokalizace ve view
- Statické lokalizace v controlleru/presenteru
- Statické lokalizace v modelu
- Dynamické lokalizace domény
- Možnost definovat způsob řešení lokalizací (soubor, databáze…)

Zhodnocení

V případě Nette je situace relativně obtížná, framework s lokalizacemi pracovat umí, ale mechanismus pro samotný překlad v základu implementován není a je třeba si ho tedy dopsat vlastními silami. Může se jednat o jakoukoliv třídu, která implementuje rozhraní ITranslator. Nabízí se tedy možnost definovat samotné úložiště dle vůle programátora. Dalším

z řešení může být stáhnutí lokalizačního rozšíření. Následující tabulka shrnuje fakta o podpoře lokalizací v obou frameworcích.

Tabulka 23 – Shrnutí možností lokalizací. Zdroj: Autor

Název	Nette	Grails
Statické lokalizace ve view	Ano	Ano
Statické lokalizace v controlleru/presenteru	Ano	Ano
Statické lokalizace v modelu	Ano	Ano
Dynamické lokalizace domény	Ne	Ne
Možnost definovat způsob řešení lokalizací (soubor, databáze…)	Ano	Ano

Grails podporuje lokalizaci pomocí souboru, ve kterém jsou překlady uchovávány formou klíč/hodnota, tedy jakýsi slovník. Tímto lze kompletně staticky lokalizovat všechny tři vrstvy. Podporu pro dynamické lokalizace standardně nenabízí. Lze ovšem, stejně jako v Nette, zvolit způsob uložení lokalizací, například prostřednictvím pluginu, který namísto properties souborů přenese lokalizace do databáze.

Dynamické lokalizace neřeší standardně ani jeden z frameworků. Domnívám se však, že je to škoda, protože nějaký jednoduchý mechanismus ve stylu CakePHP (Translate behavior[23]) by mohl být nabízen.

Tabulka 24 – Vyhodnocení kritéria podpora lokalizací. Zdroj: Autor

Nette	Grails
2-	2

3.15 Generátory

Při vývoji webových prezentací se vývojáři mnohdy potýkají s často se opakujícími úkoly, a to obzvlášť na začátku vývoje. Tento úkol však může usnadňovat generátor, který umí na základě existence nějakého objektu vygenerovat jiný. Umí založit nový projekt,

[23] Více informací o Translate behavior lze nalézt zde: http://book.cakephp.org/1.3/en/The-Manual/Core-Behaviors/Translate.html

generovat view ke controlleru, na základě databáze vygenerovat modely či další šikovné funkce, které mohou, troufám si říci, dramaticky zvýšit produktivitu. Níže jsem vyspecifikoval seznam funkcí, které by měly být standardem u každého generátoru. Tyto funkcionality jsou základem. Je samozřejmé, že generátor může nabízet podstatně širší spektrum funkcí i v závislosti na tom, co všechno v něm lze vytvářet (tagy, helpery…).

- Vytvořit projekt
- Provést základní nastavení projektu
- Vygenerovat a případně nastavit model
- Vygenerovat controller na základě modelu
- Vygenerovat view na základě controlleru
- Vygenerovat testy

Zhodnocení

Toto kritérium je velmi snadné na vyhodnocení. Ze všech definovaných je zde největší rozdíl mezi frameworky. Následující tabulka dokumentuje tento fakt na přehledu podporovaných základních funkcí.

Tabulka 25 – Přehled základních podporovaných funkcí. Zdroj: Autor

Název	Nette	Grails
Vytvořit projekt	-	Ano
Provést základní nastavení projektu	-	Ano
Vygenerovat a případně nastavit model	-	Ano
Vygenerovat controller na základě modelu	-	Ano
Vygenerovat view na základě controlleru	-	Ano
Vygenerovat testy	-	Ano

Grails má generátor a Nette nikoliv. Navíc Grails disponuje velice sofistikovanou konzolovou aplikací, která zvládá velmi široké spektrum úkolů. Jedná se zatím asi o nejlepší aplikaci tohoto typu, se kterou jsem se zatím setkal.

Tabulka 26 – Vyhodnocení kritéria generátory. Zdroj: Autor

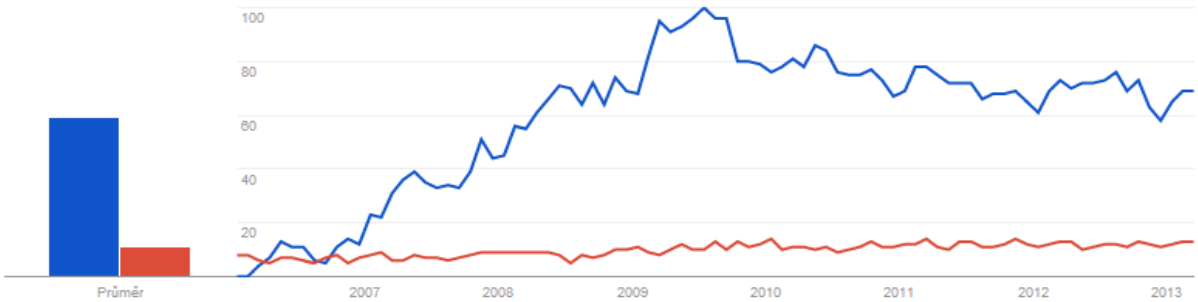
Nette	Grails
5	1

3.16 Rozšířenost a význam

Domnívám se, že vývojář při zvažování, zda začít používat či studovat daný framework, má ve svém žebříčku (samozřejmě, pokud má možnost volby) poměrně vysoko míru rozšíření daného frameworku. Je samozřejmé, že určení této hodnoty může být poměrně obtížné a nikdy se nelze dozvědět přesné číslo. Vzhledem k tomu, že nemám přístup ke specializovaným průzkumům, použiji jednoduchou metodiku hodnocení, založenou na zájmu uživatelů, vyjádřením míry hledanosti klíčových slov, které mají něco společného s řetězcem symbolizujícím název frameworku ve vyhledávači Google. Jako nejideálnější zobrazení se mi jeví vyjádření grafem vývoje hledanosti a mapou světa se zachycením, odkud daný dotaz na vyhledávač pocházel.

Zhodnocení

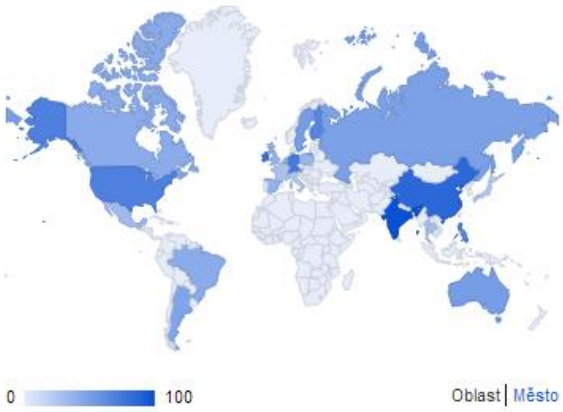
Níže uvádím graf vývoje tzv. zájmu, který vyjadřuje relativní objem vyhledávání vzhledem k nejvyššímu bodu na mapě. Tento graf naznačuje trend popularity obou frameworků od roku 2006 do roku 2013. Jako orientační měřítko ho lze brát poměrně vážně.



Obrázek 5 – Relativní zájem o Grails (modrá) a Nette (červená) v letech 2006 až 2013 dle hledanosti. Zdroj: Google

Z grafu lze vidět, že Grails mají nesrovnatelně vyšší počty vyhledávání. S největší pravděpodobností to bude způsobeno jeho celosvětovým významem, který má Nette spíše lokálnější, i když se proti tomu snaží, zatím marně, bojovat.

Další dva obrázky zobrazují tyto počty hledání dle jednotlivých států. Pomocí tohoto ukazatele lze dokladovat lokální či globální význam frameworku.



Obrázek 6 – Mapa zobrazující zdrojové státy hledání klíčového slova související s Grails. Zdroj: Google

Obrázek 7 - Mapa zobrazující zdrojové státy hledání klíčového slova související s Nette. Zdroj: Google

Mapy poměrně zřetelně naznačují, že Nette má prozatím spíše lokální význam (Česká republika) a Grails globální, zajímají se o něj lidé z celého světa. Naproti tomu statistiky, které se zaměřují na porovnání dat z České republiky, jsou značně odlišné, viz následující graf vývoje zájmu v ČR.

Obrázek 8 - Relativní zájem o Grails (modrá) a Nette (červená) v letech 2006 až 2013 dle hledanosti v ČR. Zdroj: Google

Podle grafu se s určitou pravděpodobností dá říci (data o geografické příslušnosti dotazů nejsou ve všech případech evidována), že v České republice je podstatně rozšířenější i významnější Nette.

Hodnotit rozšíření a význam tedy budu ve dvou kategoriích, a to na globální úrovni a lokální v České republice.

Tabulka 27 – Hodnocení lokálního a globálního rozšíření a významu frameworků. Zdroj: Autor

Oblast	Nette	Grails
Globální	4	1
Lokální (ČR)	1	4

Následující tabulka, která shrnuje hodnocení toho kritéria, je pouze prostým aritmetických průměrem hodnot dosažených oběma frameworky.

Tabulka 28 – Vyhodnocení kritéria rozšířenost a význam. Zdroj: Autor

Nette	Grails
2,5	2,5

3.17 Integrované vývojové prostředí

Integrované vývojové prostředí [24]představuje velice vyspělý textový editor, který má za cíl usnadnit psaní samotného kódu řadou velice užitečných funkcí, počínaje zvýrazňováním syntaxe a konče podporou pro refactoring.

Podpora samotného frameworku spočívá především ve zjednodušení práce, zvýrazňování šablon, inteligentním napovídáním, zvýrazňováním syntaxe v kódu, automatizací častých úkonů a dalšími podstatnými ulehčeními rutinní práce.

Podporu jazyka samotného považuji za standard těchto prostředí, nemá tedy význam tento fakt hodnotit.

Cílem tohoto kritéria je srovnat úroveň podpory ve třech nejpoužívanějších editorech pro oba frameworky. Naneštěstí se mi nepodařilo dostat ke zdroji, který by trh s těmito editory pro dané frameworky rozumně popsal. Zvolil jsem je tedy subjektivně dle toho, jaké editory vídám nejčastěji (domnívám se, že v případě reálného průzkumu by tyto editory byly také těmi nejčastějšími odpověďmi, protože patří dle mých zkušeností mezi nejpoužívanější obecně).

Hodnotit budu pouze základní podporu, to znamená zvýrazňování šablon (z toho důvodu, že v sobě mají velmi často nějaké nestandardní prvky), napovídání kódu

[24] Integrated Development Environment = IDE

vycházejícího z použití daného frameworku (ve všech jeho částech) a podporu generátoru kódu.

Zhodnocení

Níže uvádím tabulku, která shrnuje úroveň výše definovaných kritérií pro jednotlivé IDE a frameworky.

Tabulka 29 – Srovnání podpory základních funkcí pro práci s oběma frameworky v některých IDE. Zdroj: Autor

Kritérium	Nette			Grails		
	PHP Storm 6	Netbeans 7.3	Eclipse PDT 3.2	IntelliJ IDEA 12	NetBeans 7.3	Eclipse STS 3.8.2
Zvýrazňování syntaxe v šablonách	Ano	Ano	Ne	Ano	Ano	Ano
Napovídání kódu	Ano	Ano	Ne	Ano	Ano	Ano
Podpora generátoru kódu	-	-	-	Ano	Ano	Ano

Grails má velice dobrou podporu ve všech zmíněných IDE. Každé z těchto prostředí podporuje svým způsobem všechny uvedené funkcionality v základu. Oproti tomu Nette nemá podporu v základu nikde a pokud je nějaká podpora možná, tak je třeba ji doinstalovat pluginem (Netbeans, PHPStorm), který je v případě PHP Stormu bohužel plný chyb.

Tabulka 30 – Vyhodnocení kritéria integrované vývojové prostředí. Zdroj: Autor

Nette	Grails
2	1

3.18 Hodnocení z praktického vývoje

Toto kritérium je založené na osobních zkušenostech z praktického vývoje typové aplikace v obou frameworcích. Jedná se o velice zásadní, byť subjektivní kritérium.

Zhodnocení

Níže uvádím tabulku, která shrnuje hodnocení z vývoje aplikace. Komentáře k jednotlivým fázím je možné nalézt v kapitole Vývoj. Tentokrát každou fázi ohodnotím zvlášť a na konci uvedu prostý aritmetický průměr pro daný framework zaokrouhlený na desetiny nahoru.

Tabulka 31 – Subjektivní hodnocení z vývoje aplikace v jednotlivých etapách dle frameworků. Zdroj: Autor

Název	Nette	Grails
Instalace a zprovoznění	2	1
Adresářová struktura	1	1
Nastavení aplikace	1	1
Implementace modelů	2	1-
Implementace controllerů/presenterů	2	2
Implementace view	1	2
Implementace ostatních tříd	1	1
Implementace testů	3-	1

Hodnocení je relativně vyrovnané. V některých oblastech dominuje Nette, v jiných naopak Grails. Celkově vychází v hodnocení lépe Grails, které je podstatně rubustnější a v mnoha ohledech se s ním pracuje lépe než s Nette, a to i v případě velmi jednoduchých aplikací.

Tabulka 32 – Vyhodnocení kritéria hodnocení z praktického vývoje. Zdroj: Autor

Nette	Grails
1,6	1,3

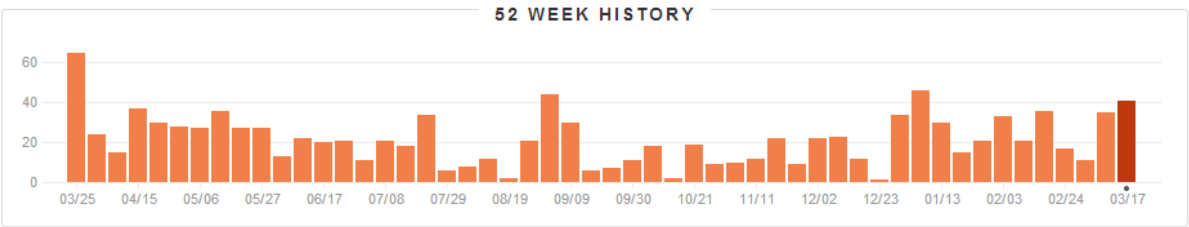
3.19 Aktivní vývoj

Aktivita samotného projektu je dalším z významných faktorů při volbě konkrétního frameworku. Aktivní vývoj neznamená pouze nové funkce či vylepšování těch stávajících, ale i opravu chyb, což je z pohledu existujícího projektu velmi důležitý aspekt.

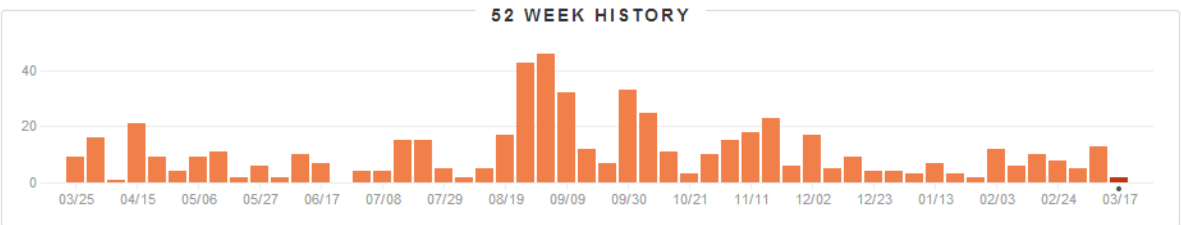
Aktivní vývoj by měl tedy být dostatečně intenzivní v opravách chyb a tedy i vydávání aktualizovaných verzí. Zároveň by mělo docházet k implementaci nových či vylepšování stávajících funkcionalit, což předpokládá existenci jakési roadmapy, tedy plánu nových funkcionalit budoucích verzí.

Zhodnocení

Analýzu aktivity frameworků jsem se rozhodl provést skrze počet commitů v Githubu, což je verzovací systém. Na základě tohoto lze s určitou spolehlivostí stanovit, zda se na daném frameworku skutečně pracuje. Níže tedy uvádím dva grafy, popisující intenzitu commitů.



Obrázek 9 – Frekvence commitů zdrojových souborů Grails. Zdroj: Github



Obrázek 10 – Frekvence commitů zdrojových souborů Nette. Zdroj: Github

Dle výše uvedených grafů lze pozorovat vyšší aktivitu Grails. Oproti tomu Nette má intenzitu commitů menší. Grails osobně považuji za rozsáhlejší projekt než Nette, a tak je větší intenzita naprosto očekávaná. Oba frameworky poměrně konstantně, pokud nepočítáme extrémy, vykazují aktivní vývoj napříč celým rokem.

Dalším bodem analýzy je frekvence vydávání minoritních [25] a majoritních[26] verzí. Cílem není větší počet verzí v co nejkratším intervalu, ale hodnota, která bude minimálně korespondovat s historickým vývojem (co se týče počtu verzí v rostoucím či stagnujícím, nikoliv klesajícím). Naneštěstí pro toto kritérium není dostatek dat, a to především časových, takže není možné toto plně vyhodnotit. Mezi majoritními verzemi Grails vycházelo v minulosti zhruba 15-20 minoritních, což nasvědčuje tomu, že na opravách se skutečně intenzivně pracuje. U Nette je situace skromnější. Mezi majoritními verzemi vyšlo 8-10 minoritních. Význam bych však tomuto faktu nepřisuzoval, jedná se o menší projekt. Počty verzí jsou napříč celou dobou vývoje frameworku konstantní a dokladuje to i fakt, že frameworky jsou aktivně vyvíjeny a opravovány.

Posledním bodem je prověření existence roadmapy, která by měla být, dle mého názoru, přítomna u každého projektu podobného rozsahu. Oba frameworky mají definovanou roadmapu a je tedy jasný vývoj, jakým se budou oba projekty ubírat.

Tabulka 33 – Vyhodnocení kritéria aktivní vývoj. Zdroj: Autor

Nette	Grails
1	1

3.20 Komunita

Kvalitu frameworku jako takového neurčují pouze autoři, ale i komunita, která se kolem něj vytvořila. Význam velké a aktivní komunity je skutečně obrovský. Díky komunitě dochází k podstatně větší informovanosti vývojářů o kvalitě (reportování chyb). Vývojáři mají zpětnou vazbu o jednotlivých částech a v neposlední řadě velmi často dochází k tomu, že se programátoři z komunity zapojují jak do vývoje frameworku samotného, tak i do rozšíření, psaní dokumentace a dalších aktivit. Přidaná hodnota může být tedy velmi významná.

Cílem tohoto kritéria není přímo porovnávat konkrétní hodnoty indikátorů, ale spíše ilustrovat aktivitu komunity na vypovídajících hodnotách. Na základě analýzy dostupných dat jsem definoval následující kritéria, která v co nejvíce objektivní míře popisují aktivitu komunity:

- Počet příspěvků na uživatele = Počet příspěvků na fóru / Počet uživatelů na fóru

[25] Malé aktualizace zdrojových kódů zaměřené především na opravy chyb
[26] Velké aktualizace přinášející nejen opravy chyb, ale i novou funkcionalitu

- Počet doplňků na uživatele = Počet doplňků vytvořených komunitou / Počet uživatelů na fóru
- Počet reportovaných bugů na uživatele = Počet reportovaných bugu celkem / Počet uživatelů na fóru
- Relativní počet uživatelů zapojených do vývoje = Počet uživatelů podílejících se na vývoji / Počet uživatelů na fóru

Indikátorů aktivity komunity by mohlo být definováno samozřejmě mnohem více, ale je nutné brát ohled na dostupnost dat. Domnívám se, že pro základní charakteristiku tato kritéria postačují.

Zhodnocení

Níže uvádím tabulku s výchozími informacemi pro výpočty.

Tabulka 34 – Výchozí hodnoty parametrů pro výpočet indikátorů aktivity. Zdroj: oficiální stránky obou frameworků

Údaj	Nette	Grails
Počet uživatelů na oficiálním fóru	5455	5281
Počet příspěvků na oficiálním fóru	90015	157747
Počet reportovaných bugů celkem	992[27]	9358[28]
Počet uživatelů z komunity podílejících se na vývoji[29]	78	100
Počet doplňků vytvořených komunitou	100	320

Z tabulky je patrné, že výsledky jednotlivých indikátorů budou velice zajímavé. Některé z hodnot jsou velmi překvapivé. Například počet uživatelů komunity zapojených do vývoje či celkový počet zaregistrovaných uživatelů na fóru, což je v případě Grails velmi zvláštní hodnota, pokud vezmeme v potaz, že framework je známý po celém světě a je i starší než Nette. Celkově hodnoty ukazují na poměrně velké zapojení komunity do vývoje, diskuze i řešení problémů.

[27] Informace o této hodnotě byly získány z Issue Trackeru na GitHubu dostupného zde https://github.com/nette/nette/issues
[28] Informace o této hodnotě byly získány z Issue Trackeru Attlasian Jira dostupného zde http://jira.grails.org/issues/?jql=project%20%3D%20GRAILS
[29] Hodnota byla získána sečtením všech uživatelů, kteří udělali alespoň jeden commit v repozitáři na Githubu

Tabulka 35 – Hodnoty indikátorů aktivity komunity. Zdroj: Autor

Údaj	Nette	Grails
Počet příspěvků na uživatele	16,5	29,8
Počet doplňků na uživatele	0,018	0,065
Počet reportovaných bugů na uživatele	0,18	1,77
Relativní počet uživatelů zapojených do vývoje	1,4%	1,8%

Z hodnot jednotlivých indikátorů by se mohlo zdát, že uživatelé v komunitě Grails více diskutují. Opak je ale pravdou. Nasbírané hodnoty z fóra Grails jsou za posledních 8 let. Stáří fóra Nette je (dle dostupných informací vyčtených ze stáří příspěvků) zhruba 4 roky. Aktivita uživatelů Nette je tedy na velmi dobré úrovni, stejně tak i Grails.

V případě doplňků je situace zřejmě podobná. Grails funguje delší dobu a bude mít tedy zřejmě více doplňků. Zde se však dle čísel domnívám, že uživatelé Grails jsou skutečně produktivnější nebo sdílnější a publikují více doplňků.

V případě bugů je situace obdobná jako u předchozího indikátoru, opět je tu znát delší doba existence frameworku. Grails má reportovaných více bugů. Vyzdvihl bych fakt, že se uživatelé do procesu hlášení chyb aktivně zapojují, protože bez toho by frameworky s největší pravděpodobností nebyly zdaleka tak odladěné.

Počet uživatelů zapojených do vývoje je relativně podobný, hodnota tohoto indikátoru tedy není příliš zajímavá. Obě hodnoty ukazují tedy na jakýsi standard.

Dle výsledků se domnívám, že aktivita komunity je na obou stranách na velmi dobré úrovni.

Tabulka 36 – Vyhodnocení kritéria aktivní komunita. Zdroj: Autor

Nette	Grails
1	1

3.21 Dokumentace

Kvalitní dokumentaci lze považovat za základ každé veřejně dostupně aplikace. V případě frameworků se lze však velice často setkat se špatnou dokumentací. Tento fakt

velice znesnadňuje vývoj a nutí vývojáře experimentovat, ztrácet drahocenný čas vyhledáváním informací, které by měly být rychle dostupné.

Toto kritérium nelze hodnotit v žádném případě objektivně, vždy bude mít subjektivní ráz. Objektivní měření pokrytí dokumentovaných částí frameworku nelze správně vyhodnotit, protože pokrytí neznamená, že úroveň popisu je dostatečně návodná, vhodná či správně zpracovaná.

Úroveň dokumentace v žádném případě nezahrnuje publikované příspěvky na diskuzních fórech. Tímto bych však nerad snížil jejich váhu, která je více než významná.

Zhodnocení

Grails má velice dobře zpracovanou dokumentaci. Až na několik výjimek se mi podařilo najít při praktickém vývoji vždy v podstatě všechno, co jsem potřeboval. Dokumentace je velmi přehledná a zpracovaná v anglickém jazyce, tudíž by žádný pokrokovější uživatel počítače neměl mít problém se zorientovat.

Situace v Nette je podstatně složitější. Existují dvě verze dokumentace, česká a anglická. Obě dvě jsou nedostatečně zpracované, uživatel zde najde opravdu málo informací a musí se až příliš často obracet na fórum a hledat v historii i poměrně základní informace. Anglická a česká verze se přitom liší, česká je na tom o něco lépe s aktuálností obsahu.

Tabulka 37 – Vyhodnocení kritéria dokumentace. Zdroj: Autor

Nette	Grails
2-	1

3.22 Pracovní nabídky

Při výběru frameworku se také vyplatí částečně orientovat podle poptávky. Vysoká odbornost v daném frameworku je sice výborná, ale v případě malých možností uplatnění bude vývojáře spíše od přechodu na danou platformu odrazovat.

Pracovní nabídky budu srovnávat opět na lokální a globální úrovni, která bude s největší pravděpodobností velmi odlišná. Hodnoty o počtech pracovních nabídek budu získávat z pracovních portálů.

Zhodnocení

Na lokální úrovni využiji data z pracovního portálu Jobs.cz a Monster.cz, na globální pak Indeed.com a americký portál Monster.com. Níže uvedená tabulka shrnuje výsledky na klíčová slova Nette a Grails.

Tabulka 38 – Počty nabídek zaměstnání vybraných pracovních portálů k 24.3.2013. Zdroj: Autor dle dat z daných pracovních portálů

Pracovní portál	Nette	Grails
Lokální úroveň		
Jobs.cz	17	1
Monster.cz	4	0
Celkem	**21**	**1**
Globální úroveň		
Indeed	0	868
Monster.com	0	11
Celkem	**0**	**879**

Již podle kritéria rozšířenosti a významu jsem předpokládal, že pracovní uplatnění Nette v cizině bude problematické. Na obou zahraničních serverech byla poptávka po vývojářích v Nette nulová. Tento fakt opět dokumentuje význam Nette pouze na úrovni České republiky. Situace na lokální úrovni je o něco lepší oproti Grails, který bývá dle nabídek velmi často volen jako doplňkový požadavek k Java EE, Springu či dalším Java technologiím. Na globální úrovni vede s naprostým přehledem Grails, který je hojně poptáván po celém světě.

Tabulka 39 – Vyhodnocení kritéria pracovní nabídky. Zdroj: Autor

Nette	Grails
2	2

3.23 Srovnání frameworků

Důležitost jednotlivých kritérií jsem stanovil metodou Fullerova trojúhelníku (jinak řečeno metodou párového srovnávání) a na základě tohoto postupu jsem tedy vypočítal váhy pro jednotlivá kritéria. Celý výpočet je z důvodu své rozsáhlosti uveden v příloze č. 2, níže uvádím výslednou tabulku kritérií s vypočítanými vahami a konkrétní hodnoty obou frameworků.

Tabulka 40 – Závěrečná sumarizační tabulka všech kritérií. Zdroj: Autor

Kritérium	Váha	Nette	Nette s váhou[30]	Grails	Grails s váhou[31]
Dostupnost	0,9%	1	0,9	1	0,9
Podpora	0,4%	5	2	2	0,8
Architektura frameworku	6,5%	1	6,5	1	6,5
Práce s adresami	5,6%	1	5,6	1	5,6
Convention over Configuration	3,0%	1	3	1	3
Dependency Injection	3,0%	3	9	1	3
Podpora testování	3,0%	3	9	1	3
Integrované funkce v základu	3,5%	1	3,5	2	7
Dostupné běhové prostředí	0,9%	2	1,8	1	0,9
AJAX/AJAJ	3,5%	1	3,5	2	7
Persistentní a doménová vrstva	7,8%	2	15,6	1	7,8
Šablonování	6,9%	1	6,9	2	13,8
Možnosti rozšíření	5,6%	3	16,8	1	5,6

[30] Hodnota je pro lepší čitelnost vynásobená 100
[31] Hodnota je pro lepší čitelnost vynásobená 100

Kritérium	Váha	Nette	Nette s váhou[30]	Grails	Grails s váhou[31]
Podpora lokalizací	3,9%	2,5	9,75	2	7,8
Generátory	6,1%	5	30,5	1	6,1
Rozšířenost a význam	4,3%	2,5	10,75	2,5	10,75
Integrované vývojové prostředí	1,7%	2	3,4	1	1,7
Hodnocení z praktického vývoje	9,1%	1,6	14,56	1,3	11,83
Aktivní vývoj	8,2%	1	8,2	1	8,2
Komunita	7,4%	1	7,4	1	7,4
Dokumentace	8,2%	2,5	20,5	1	8,2
Pracovní nabídky	0,4%	2	0,8	2	0,8
Celkem	**100%**	**2,05[32]**	**189,96[33]**	**1,35[34]**	**127,68[35]**

Vítězně vyšel z porovnání framework Grails, lepší hodnota byla v tomto případě ta nižší. Domnívám se, že vítězství je oprávněné a to z několika důvodů. Framework je skutečně propracovaný, má již v základu pokrytou celou řadu důležitých funkcionalit, jako je například ORM nebo generátor kódu. Tyto a další aspekty dělají z Grails velice komplexní nástroj pro rychlý vývoj webových stránek.

Nette má v tomto srovnání lehký hendikep v podobě kratší doby celkového vývoje, což se podepisuje na existenci nebo kvalitě některých základních prvků typu generátor, dokumentace nebo ORM (které není ani v plánu). I přesto framework dokázal dominovat v části zaměřující se na šablonování nebo AJAX, což jsou velice silné stránky Nette.

Největší rozdíly mezi oběma frameworky jsou především v Dependency Injection kontajneru, podpoře testování, podpoře AJAXu, dokumentaci, možnostech rozšíření a samozřejmě v generátoru. Rozdíl je samozřejmě i v persistentní a doménové vrstvě, avšak díky zaměření obou frameworků tento rozdíl nepovažuji za porovnávatelný (Nette chce

[32] Průměr z hodnot
[33] Suma hodnot, čím menší, tím lepší
[34] Průměr z hodnot
[35] Suma hodnot, čím menší, tím lepší

poskytovat jednoduchou vrstvu naproti tomu Grails komplexní vrstvu), takže hodnocení s přihlédnutím ke strategii v daném frameworku mají oba frameworky podobné. Zajímavý je také fakt, že celkové hodnocení obou frameworků relativně dobře koresponduje s hodnocením z praktického vývoje.

Oba frameworky jsou relativně vyzrálé a bez jakýchkoliv obav bych využil oba dva na komerčním projektu. Z povahy frameworku si troufám tvrdit, že Grails má podstatně navrch při složitějších úlohách. Ve standardních CRUD úlohách bude situace vyrovnanější, pouze lehce nakloněná na stranu Grails (hlavně díky generátoru). Výběr je pouze na preferencích či zkušenostech daného vývojáře nebo týmu. Oba dva frameworky nabízí solidní základ pro rychlý vývoj aplikací.

4 Návrh aplikace

Závěrečnou fází srovnání frameworků bude vývoj jedné konkrétní aplikace. Nejprve je nutné tuto aplikaci dostatečně podrobně popsat a navrhnout. Aplikace bude pro obě platformy identická, tudíž bude zajímavé pozorovat, jakým způsobem probíhá vývoj v obou prostředích.

Nutno však podotknout, že primárním cílem této práce není návrh a vývoj samotné aplikace. Nebude zde tedy kladen důraz na vznik dokonalé aplikace skrze perfektní zdrojový kód, nýbrž spíše o demonstraci přístupů/možností obou frameworků při vývoji standardních[36] aplikací využívajících pouze funkce dostupné v základu.

4.1 Popis aplikace

V této kapitole kompletně popíši požadovanou funkčnost aplikace na dostatečné úrovni pro návrh a vývoj. Aplikaci bylo nutné navrhnout takovým způsobem, aby přítomnost frameworku při vývoji byla plusem, nikoliv mínusem. V žádném případě by se nemělo jednat o aplikaci, která je velmi silně individuální (téměř framework nevyužívá), příliš malá (framework je v tomto případě „kanón na vrabce“) či je příliš velká/rozsáhlá (RAD framework nemusí být to nejlepší řešení).

4.1.1 Účel aplikace

Cílem bude vyvinout aplikaci na organizaci úkolů, která bude částečně obsahově čerpat z metody organizace práce GTD.[37] Nyní pouze velice stručně pár slov k metodice. Jejím cílem je efektivní hospodaření s časem tak, aby člověk zvládal úkoly, které má, ale zároveň nebyl ve stresu. K tomu faktu má dle [Allen, 2002] pomoci tvorba sofistikovaných seznamů, které s tímto problémem pomohou. Samozřejmě nestačí jen seznamy, ale je nutné začít myslet a procesně fungovat tak, jak to popisuje metodika. V opačném případě seznamy ztrácí význam.

[36] Za standardní považuji aplikace mající převážně CRUD charakter
[37] Velice známá matodika Getting Things Done (Mít vše hotovo) od amerického autora Davida Allena. Více informací lze nalézt v tomto zdroji: [Allen, 2002]

4.1.2 Funkční specifikace

Aplikace by měla poskytovat prostřednictvím webového formuláře rozhraní pro registraci a dále pak možnost přihlašování/odhlašování uživatelů. Ústředním prvkem by měl být přehled všech úkolů, které byly do systému vloženy. Tyto úkoly by měly obsahovat název, popis, prioritu, datum do kdy a stav. Stavem je myšleno, zda je úkol aktivní, čeká, bude někdy realizován, anebo je hotov. Ke každému úkolu by mělo být možné si psát průběžně komentáře.

Každý úkol by měl mít možnost přiřazení jedné či více značek (tagů). Posléze v obecném přehledu dle těchto značek půjde filtrovat.

Všechny úkoly by se měly dát seřadit dle názvu, priority, data do kdy a aktuálního stavu. Aplikace by měla též poskytovat možnost úkoly filtrovat dle atributu do kdy, stavu a samozřejmě vyhledávání dle názvu.

Celkově by aplikace měla poskytovat velice přehledné, rychlé a intuitivní rozhraní bez přebytečné grafiky.

4.1.3 Předpoklady a omezení

Předpokládá se využití běžných praktik, kdy URL adresy jsou v pěkném tvaru a je využíváno sémanticky správných značek v HTML, je bezpečná (proti běžným útokům typu SQL Injection,…) a další aspekty, které považuji za samozřejmost, a jejichž explicitní vydefinování není v tomto textu účelné.

Aplikace bude kompletně realizována, ale nedoporučuji její okamžité nasazení do ostrého provozu bez řádného otestování na simulovaném produkčním prostředí.

4.2 Návrh aplikace

Provedu návrh aplikace na globální i detailní úrovni (podobně jako [Odehnal, 2011]). Globální úroveň definuje aplikaci na technologické úrovni[38], detailní pak na implementační úrovni[39] dle obou frameworků.

[38] Nezávislé na implementačním prostředí = jazyk a framework
[39] Závislé na implementačním prostředí = jazyk a framework

Při návrhu aplikace na obecné úrovni budu vycházet z postupu, který definovala [Buchalcevová, a další, 2007]. Pro potřeby a charakter této práce (není třeba přílišná úroveň detailu) ho však upravím:

a) Popis případů užití

b) Návrh tříd a jejich odpovědností

c) Vytvoření statického diagramu tříd

d) Systémový návrh

e) Návrh uživatelského rozhraní

Detailní úroveň bude spočívat prakticky pouze v definování diagramu tříd aplikace spolu s pohledy.

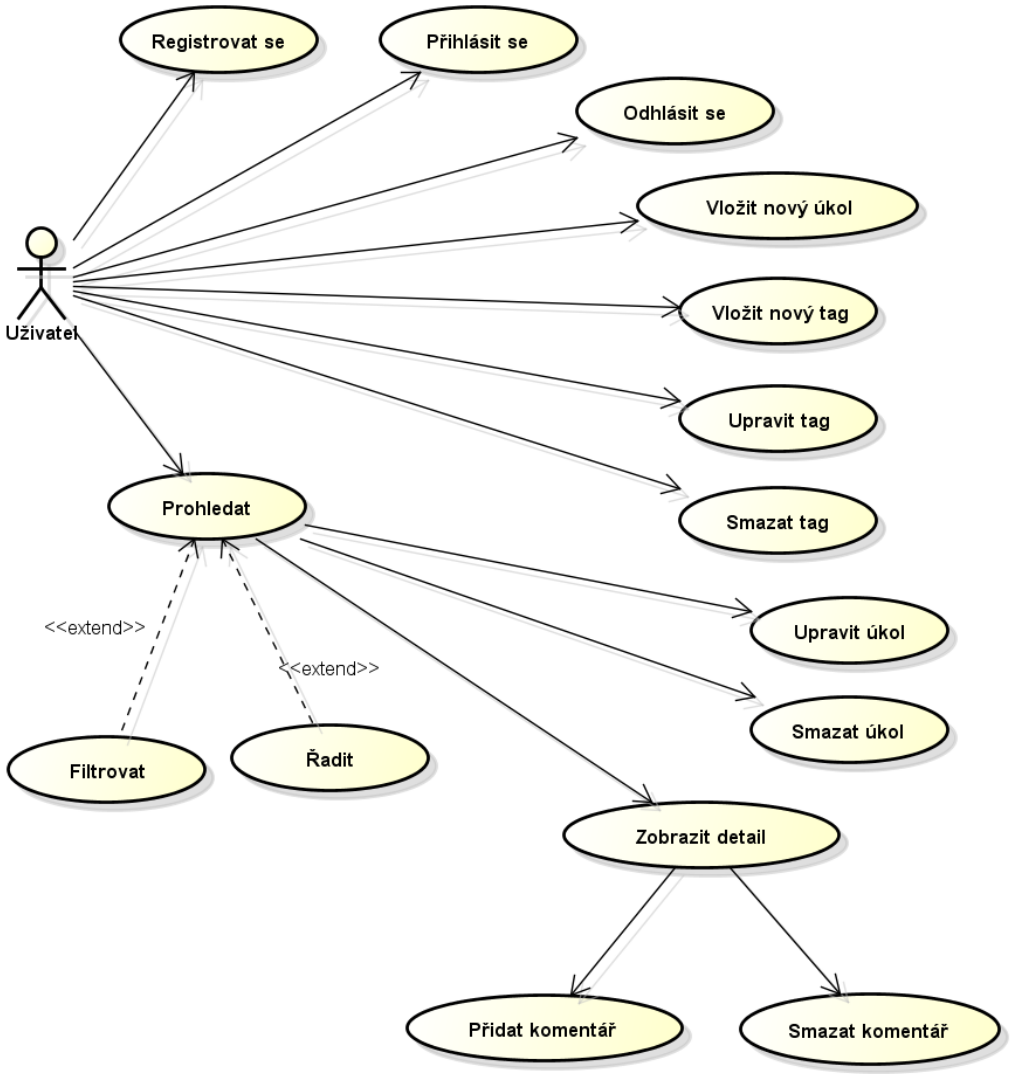
4.3 Globální návrh aplikace

V této kapitole navrhnu aplikaci na takové obecné úrovni, která bude společná oběma frameworkům. Bude se jednat především o popis případů užití, návrh tříd, systémový návrh a návrh uživatelského rozhraní aplikace.

4.3.1 Popis případů užití

Dle funkční specifikace jsem identifikoval čtrnáct základních případů užití. Obecně se dají rozdělit na autentifikaci, správu profilu, správu úkolů, práci s komentáři a vyhledávání.

Obrázek 11 představuje diagram případů užití. Pod obrázkem následuje stručný popis všech případů užití, návrh tříd, systémový návrh a návrh uživatelského rozhraní. Níže uvádím diagram a ilustrativně několik popisů případů užití. Ostatní jsou k dispozici v příloze č. 1.



Obrázek 11 – Diagram případů užití. Zdroj: Autor

Tabulka 41 – Případ užití – Registrace uživatele. Zdroj: Autor

Název případu užití	Registrace uživatele
Identifikace	UC1
Cíl	Zaregistrovaný uživatel
Primární aktér	Uživatel
Vstupní podmínky	Uživatel není přihlášen
Výstupní podmínky	Uživatel byl zaregistrován
Základní scénář	

Krok	**Role**	**Akce**
1	Uživatel	Klikne na tlačítko registrovat nového uživatele

	2	Aplikace	Zobrazí varovnou hlášku a přesměruje na úvodní stranu v případě, že nebyly dodrženy vstupní podmínky. V opačném případě zobrazí registrační formulář.
	3	Uživatel	Vyplní hodnoty
	4	Aplikace	Zkontroluje vyplněné hodnoty, pokud jsou chybné, zobrazí chybovou hlášku a zobrazí vyplněný formulář s označenými chybnými políčky. V opačném případě zaregistruje uživatele, přesměruju na hlavní stranu a zobrazí hlášku, že se registrace zdařila.

Tabulka 42 – Případ užití – Přihlášení uživatele. Zdroj: Autor

Název případu užití	Přihlášení uživatele		
Identifikace	UC2		
Cíl	Přihlášený uživatel		
Primární aktér	Uživatel		
Vstupní podmínky	Uživatel není přihlášen		
Výstupní podmínky	Uživatel je přihlášen		
Základní scénář	**Krok**	**Role**	**Akce**
	1	Uživatel	Na úvodní straně vyplní přihlašovací údaje a klikne na tlačítko odeslat.
	2	Aplikace	V případě, že validátor odhalí neplatné hodnoty, přesměruje na hlavní stranu a zobrazí hlášku o chybách. V opačném případě, pokud hodnoty v databázi existují, přihlásí uživatele a přesměruje na stránku s přehledem úkolů.

4.3.2 Návrh tříd a jejich odpovědností

Nyní, když jsem vydefinoval a popsal případy užití, je možné identifikovat jednotlivé třídy business logiky a popsat je. Aplikační a zobrazovací logikou se budu zabývat v detailním návrhu.

Jako popisnou metodu jsem si vybral dle [Buchalcevová, a další, 2007] zjednodušenou CRC kartu, která identifikuje nejen třídu samotnou, ale i její odpovědnosti a závislosti na jiných třídách (spolupracovníci). Atributy předka a potomků vynechávám, protože v mém případě nemají opodstatnění. Níže jsou k jednotlivým identifikovaným (doménovým[40]) třídám uvedeny CRC karty formou tabulek.

Za názvem budu v závorkách uvádět anglické ekvivalenty, které budu v pozdější fázi používat namísto českých, především z důvodu konvencí frameworků, ale i jazykové jednolitosti napříč celou aplikací.

Tabulka 43 – CRC karta - Uživatel. Zdroj: Autor

Uživatel (User)	
Odpovědnosti	**Spolupracovníci**
Údaje o uživateli včetně přihlašovacích údajů Ověření platnosti přihlašovacích údajů Vytvoření nového záznamu	

Tabulka 44 – CRC karta - Úkol. Zdroj: Autor

Úkol (Task)	
Odpovědnosti	**Spolupracovníci**
Údaje o úkolu Vytvoření nového záznamu Úprava záznamu	Stav úkolu Značka Uživatel

[40] Lépe řečeno doménový model (Domain Model), který obsahuje jak data samotná data, tak chování. [Fowler, a další, 2002]

Úkol (Task)	
Smazání záznamu Multikriteriální vyhledávání se stránkováním a řazením Řazení úkolu	Komentář

Tabulka 45 – CRC karta – Stav úkolu. Zdroj: Autor

Stav úkolu (State)	
Odpovědnosti	**Spolupracovníci**
Název stavu	Úkol

Tabulka 46 - CRC karta - Značka. Zdroj: Autor

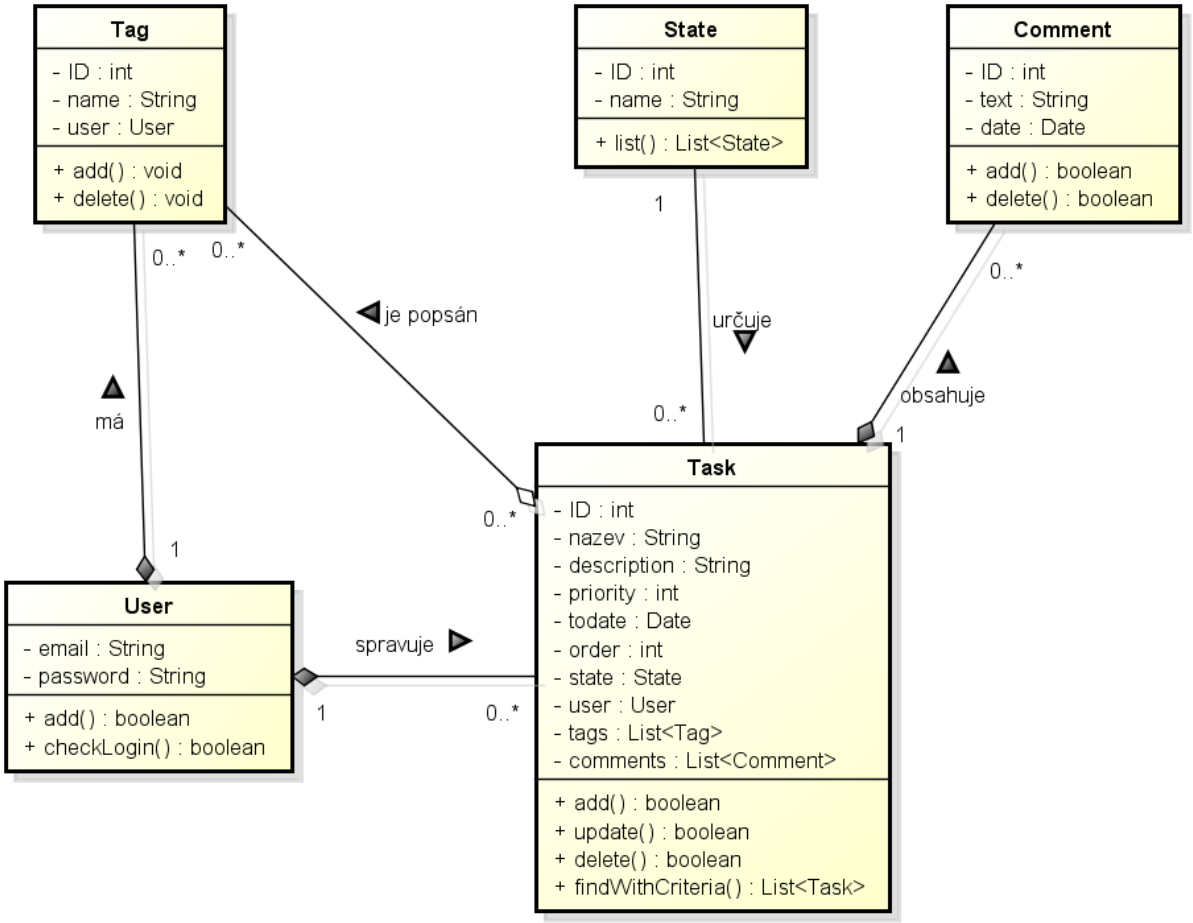
Značka (Tag)	
Odpovědnosti	**Spolupracovníci**
Název značky	Úkol

Tabulka 47 - CRC karta - Komentář. Zdroj: Autor

Komentář (Comment)	
Odpovědnosti	**Spolupracovníci**
Text komentáře	Úkol

4.3.3 Vytvoření statického diagramu tříd

Na základě identifikovaných tříd a jejich vlastností v předchozí kapitole je možné navrhnout diagram tříd, který abstrahuje od implementačních specifik. V samotné aplikaci poslouží pro návrh databázové struktury a definování modelů (objektů zapouzdřujících business logiku).



Obrázek 12 – Statický diagram tříd. Zdroj: Autor

4.3.4 Systémový návrh

V této kapitole podrobněji vydefinuji implementační prostředí aplikace v závislosti na variantě. Veškeré produkty třetích stran budou volně šířitelné.

Společným pro obě varianty aplikace bude operační systém, kterým bude Ubuntu Linux 12.10 Server 64bit[41], který využívá osvědčené jádro distribuce Debian. Další společnou částí bude databázový systém Oracle MySQL 5.5.28 [42] v komunitní verzi dostupný zdarma.

Obě varianty aplikace budou mít formu webových stránek.

[41] Dostupný zde: http://www.ubuntu.com/download/server
[42] Dostupný zde: http://www.mysql.com/downloads/

Framework Nette

Webový server pro běh PHP aplikací bude Apache 2.2[43]. Programovací jazyk je dle frameworku PHP[44] (ve verzi 5.4, pro Nette 2 je nutné alespoň 5.2). Samotný Framework Nette [45]bude použit ve verzi 2.0.8. Nastavení platformy je standardní, u Apache by možná mohlo být považováno za nestandardní aktivování modulu mod_rewrite. (nastavení není nutné, ale pro plnou funkčnost doporučené [46])

Framework Grails

V případě frameworku Grails bude pro běh aplikace použit webový kontajner Apache Tomcat 7.0.33[47] (pro spouštění aplikace při vývoji pod Apachem, namísto výchozího Jetty, postačí doinstalovat plugin[48]). Programovací jazyk je dle frameworku Groovy 2.0 a samotný framework Grails[49] ve verzi 2.1.1. (balík s Grailsy obsahuje i jazyk Groovy). Pro běh Groovy a Grails je nutné JDK[50] ve verzi minimálně 1.6.

4.3.5 Návrh uživatelského rozhraní

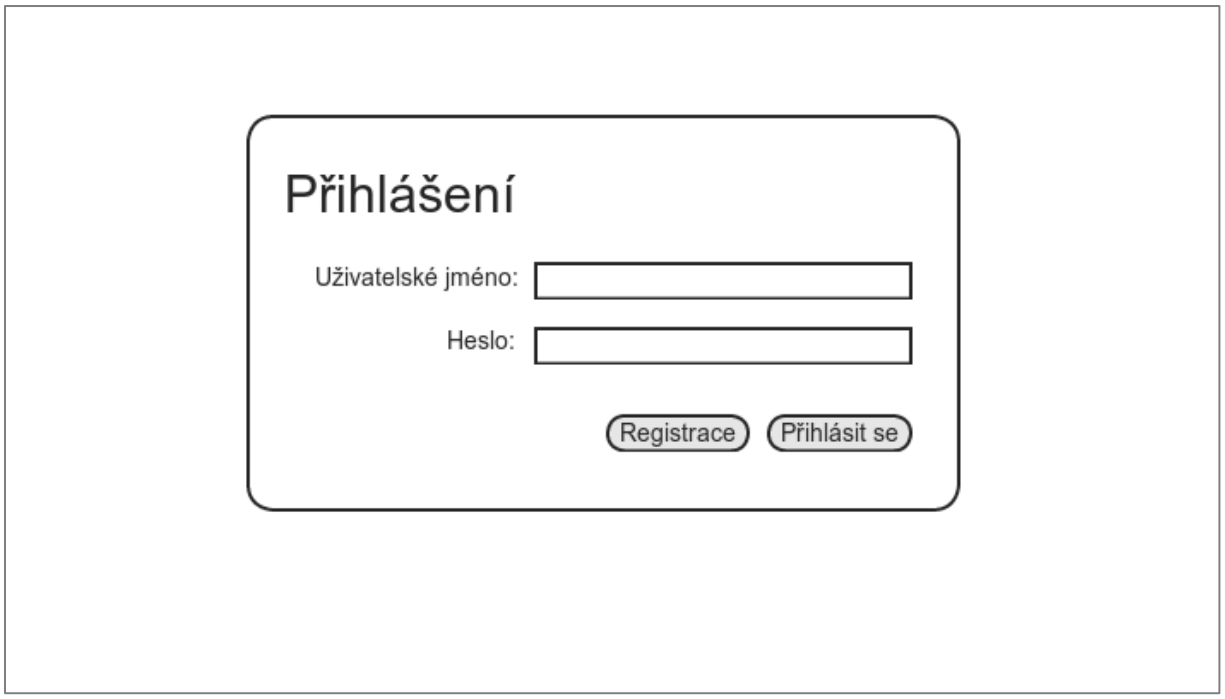
Pro návrh tzv. GUI[51] jsem využil wireframy, což je schématický drátový model kreslený v tomto případě jakoby „od ruky“. Odpadá tedy nutnost prototypovat na papír. V softwaru (například Mockingbird[52]) je návrh z mého pohledu podstatně pohodlnější a přehlednější. Samozřejmě pouze v případě, že se jedná o skutečně kvalitní software.

Význam návrhu grafického rozhraní předem spočívá především v pečlivém rozmístění všech uživatelských prvků s velice jednoduchou úpravou (což v případě, že je návrh již implementován, příliš snadné být nemusí). Tento návrh lze jednoduše nechávat hodnotit od uživatelů a posléze optimalizovat bez nutnosti mít vyvinutý funkční prototyp aplikace.

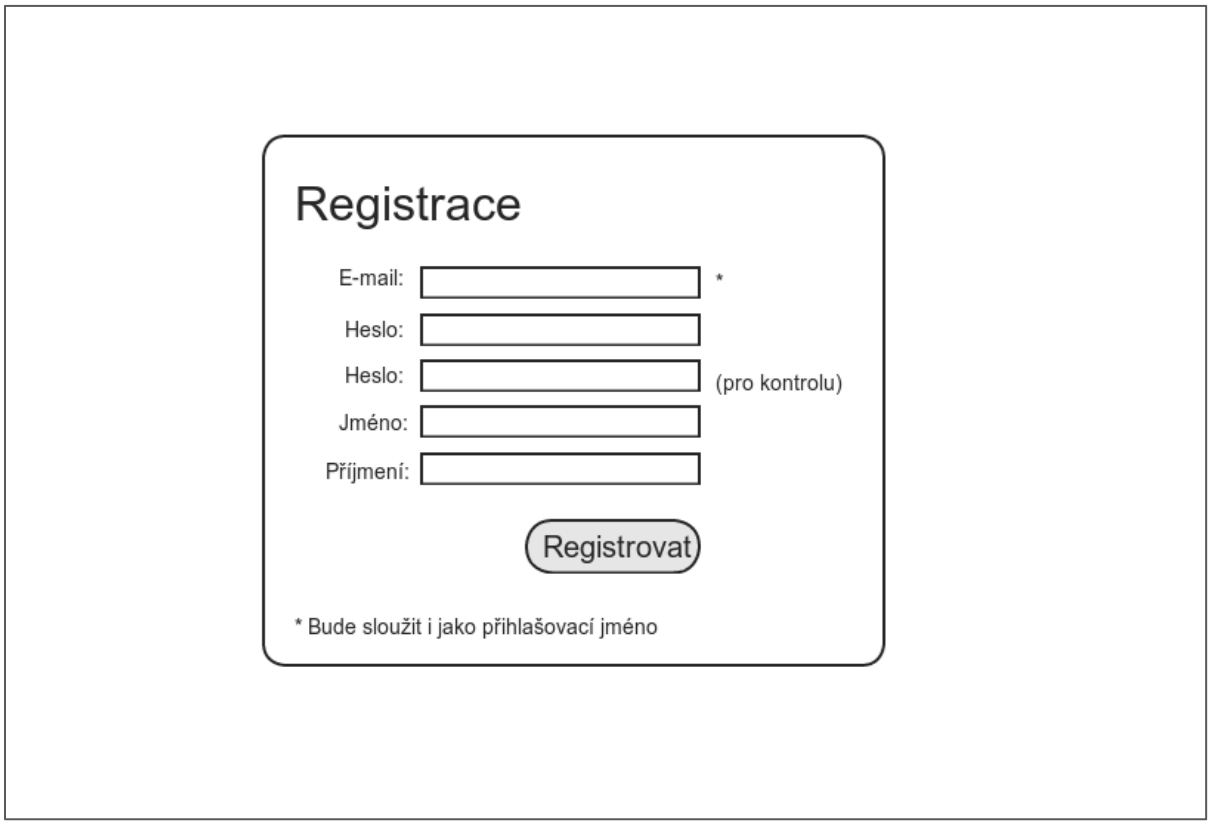
Níže jsem na základě funkční specifikace a případů užití vytvořil tři wireframy základních obrazovek. První je přihlášení, následuje registrace a hlavní strana. Význam má tento návrh spíše ve smyslu rozvržení jednotlivých prvků.

---

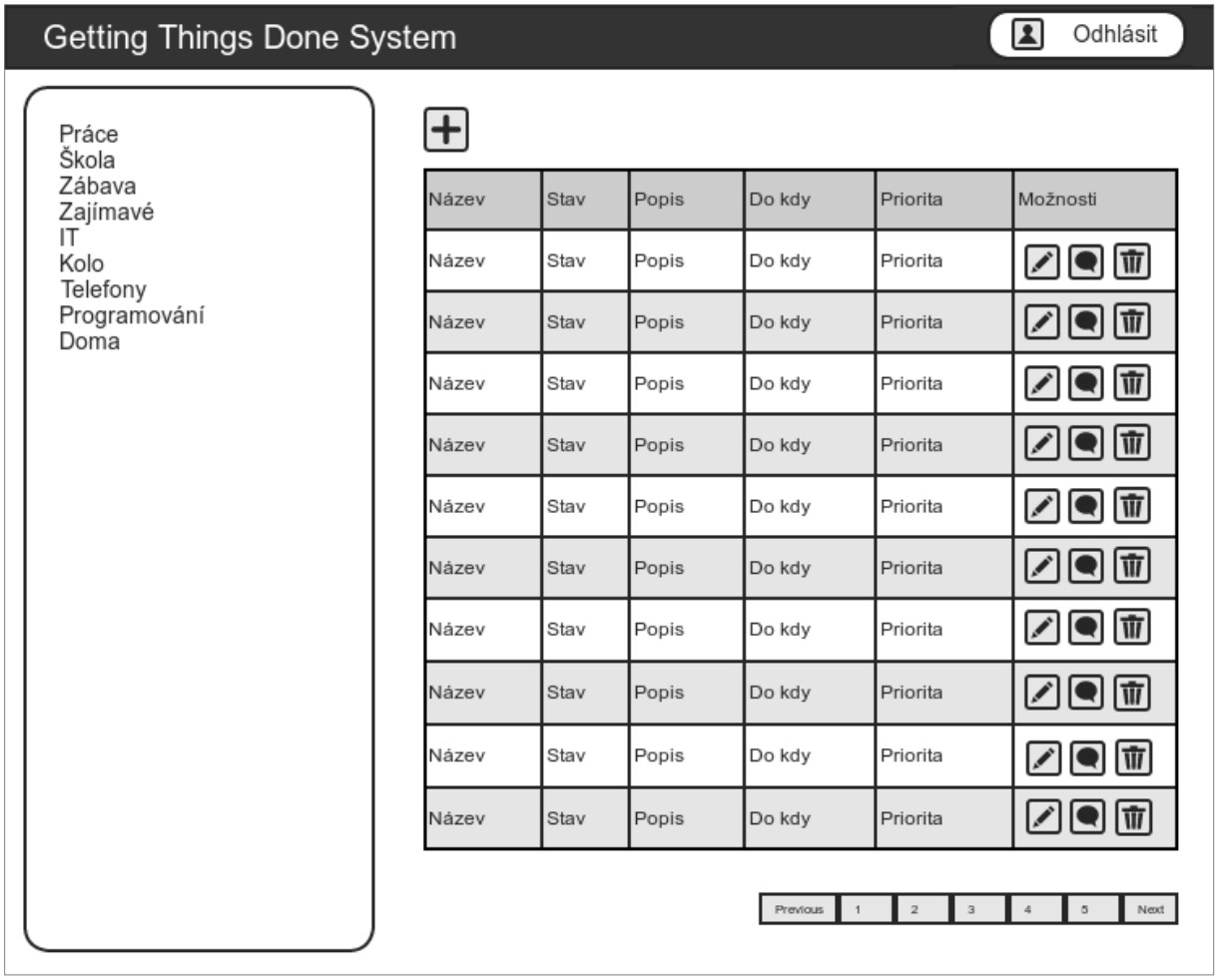
[43] Dostupný zde: http://httpd.apache.org/download.cgi
[44] Dostupný zde: http://php.net/downloads.php
[45] Dostupný zde: http://nette.org/cs/download
[46]Více informací o požadavcích frameworku zde: http://doc.nette.org/cs/requirements
[47] Dostupný zde: http://tomcat.apache.org/download-70.cgi
[48] Návod na instalaci je dostupný zde: http://grails.org/plugin/tomcat
[49] Dostupný zde: http://grails.org/download
[50] Java Development Kit – Dostupný zde: http://www.oracle.com/technetwork/java/javase/downloads/index.html
[51] Graphical User Interface = grafické uživatelské rozhraní
[52] Dostupný zde: https://gomockingbird.com/



Obrázek 13 – Wireframe - Přihlášení. Zdroj: Autor



Obrázek 14 – Wireframe - Registrace. Zdroj: Autor



Obrázek 15 – Wireframe – Hlavní strana. Zdroj: Autor

4.4 Detailní návrh aplikace

V této kapitole navrhnu aplikaci na implementační úrovni, tj. se specifiky jednotlivých frameworků. Nepovažuji za podstatné pro účel této práce definovat na základě statického diagramu entitně-relační schéma pro databázovou strukturu (opis kódu pro vygenerování schématu bude však přiložen). Dále se budu zabývat architekturou aplikace pro konkrétní framework.

Detailní návrhy v obou frameworcích se liší od obecné předlohy, rozdíly u každého zvlášť popíši. Zelenou barvou je naznačena část Model, modrou Controller a oranžovou View (jednotlivé šablony a layout).

Obrázek 16 zobrazuje architekturu ve frameworku Grails. Rozdíl oproti globálnímu návrhu spočívá v absenci metod u modelu, které nebylo nutné explicitně definovat znovu,

protože základní operace s doménovou třídou jsou automaticky k dispozici z ORM frameworku GORM.

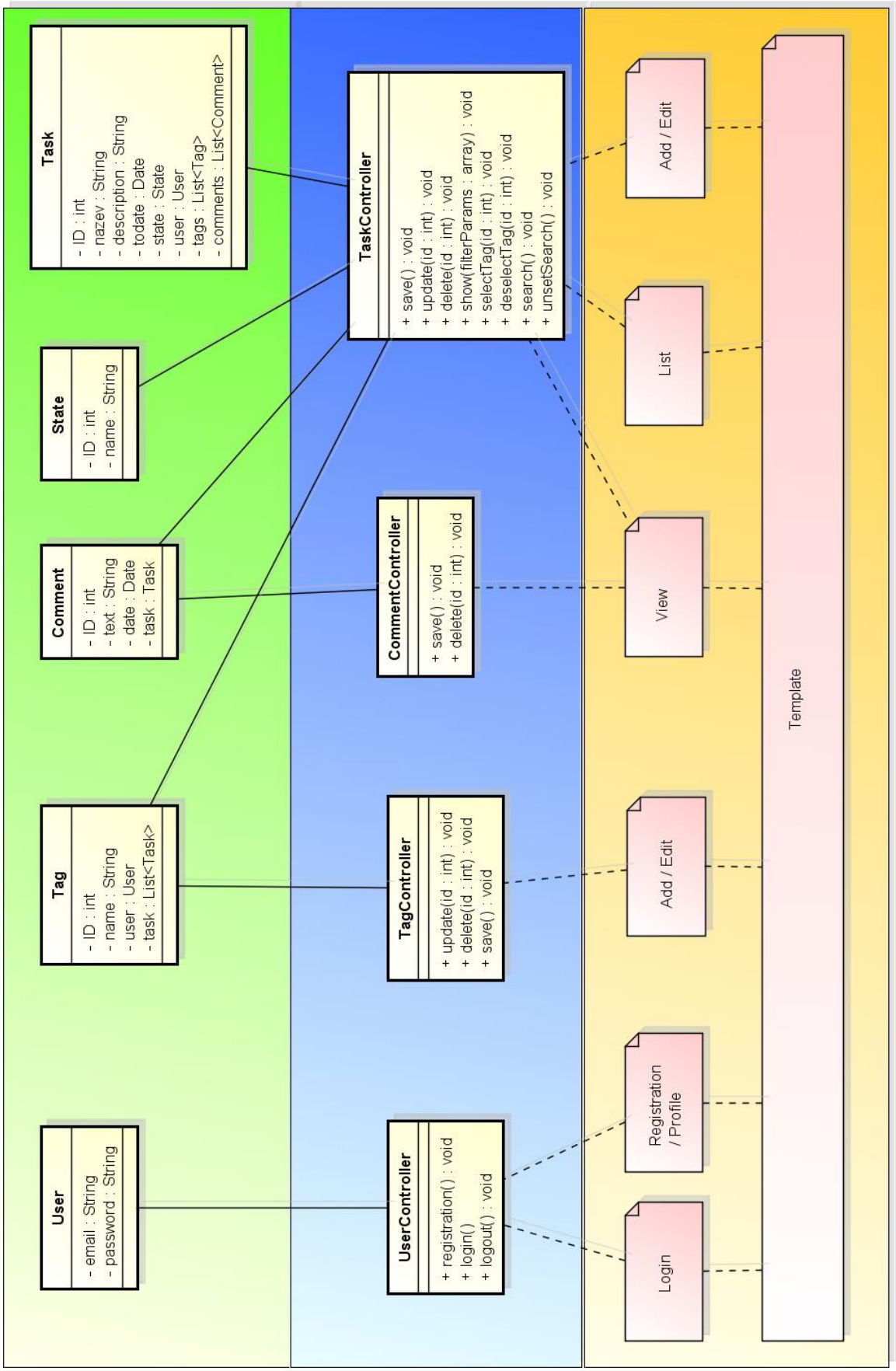
Obrázek 17 zachycuje architekturu ve frameworku Nette. Díky tomu, že Nette nevyužívá ORM vrstvu, ale pouze částečně implementovanou knihovnu NotORM, modely slouží jen jako jakési velmi zjednodušené rozhraní pro přístup k datům z tabulek. Díky těmto faktům model nepotřebuje mít definované atributy, ale pouze metody pro práci s modelem, resp. přímo s tabulkou (zde je velký rozdíl oproti Grails). Controllery se zde jmenují, díky odlišnému chápání jejich účelu, presentery. I přes to, že vypadají velmi podobně a mohlo by se zdát, že jsou stejné, budou se lišit v práci s Modelem a View.

V podstatě shodné jsou části Controller a View. V Controlleru jsou nadefinovány základní akce:

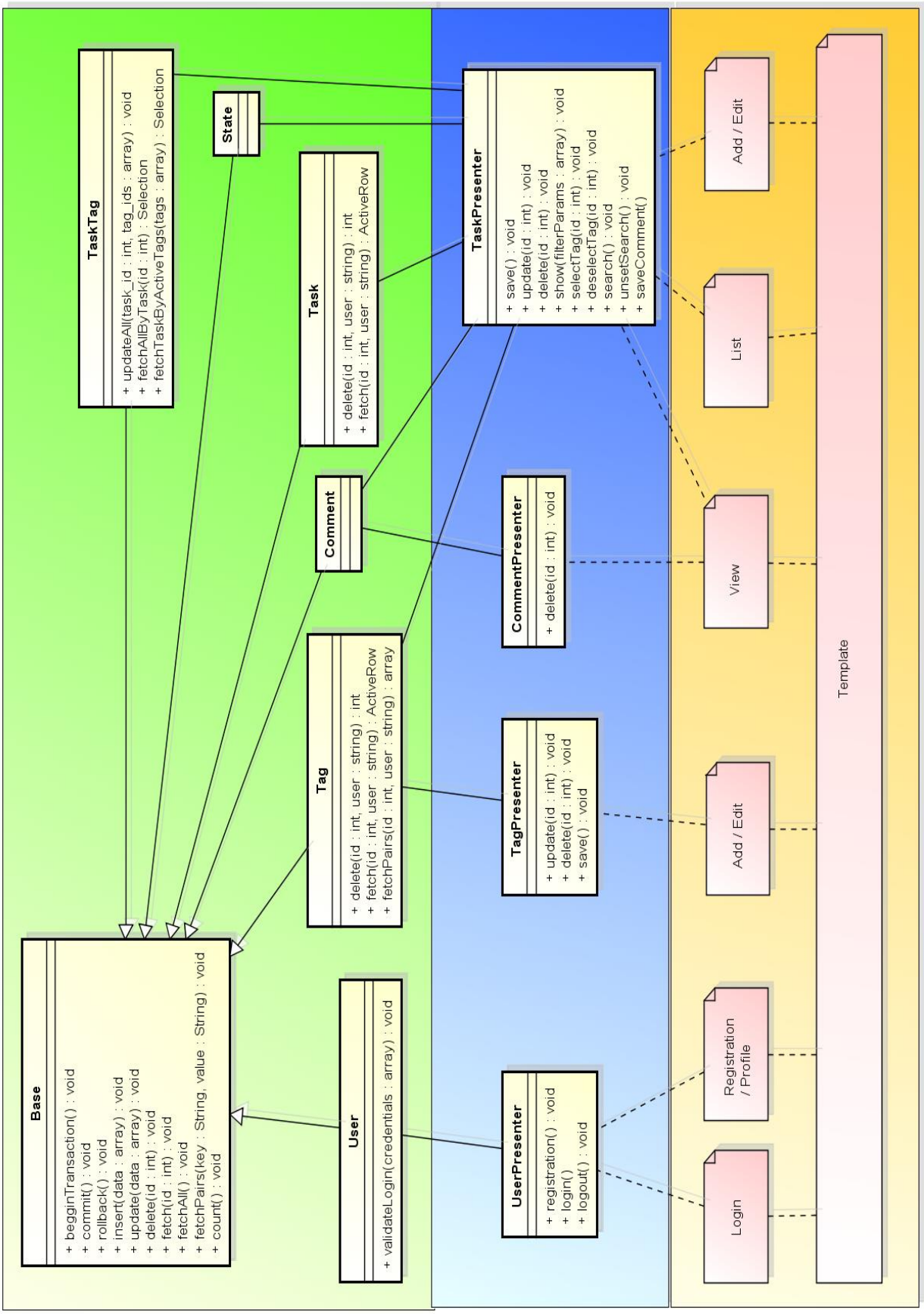
- Uživatel
    - Registrace
    - Přihlášení
    - Odhlášení
- Komentář
    - Uložit
    - Smazat
- Úkol
    - Uložit
    - Aktualizovat
    - Smazat
    - Zobrazit
    - Seřadit a filtrovat
- Tag
    - Uložit
    - Aktualizovat
    - Smazat

Domnívám se, že slovní popis jednotlivých akcí není nutný a v podstatě vychází ze statického modelu tříd.View jsou zobrazována v jedné globální specifické šabloně (obecně se jí říká Layout). Každé jednotlivé View odpovídá akcím v Controllerech, pokud jsou potřeba.

Díky architektonickému vzoru MVC, resp. MPV jsou diagramy velmi podobné.



Obrázek 16 – Architektura aplikace ve frameworku Grails. Zdroj: Autor



Obrázek 17 – Architektura aplikace ve frameworku Nette. Zdroj: Autor

5 Vývoj

V této kapitole popíšu, jakým způsobem probíhal vývoj aplikace v obou frameworcích s názornou ukázkou fragmentů kódů pro ilustrování rozdílů. Cílem této práce není návrh šablon, tudíž je v popisech vynechám. Záměrně se budu snažit prezentační vrstvu modifikovat minimálně od výchozí podoby z toho důvodu, aby bylo vidět, jaké styly a šablony používá framework ve výchozím nastavení. Zároveň se nedomnívám, že je nutné či účelné uvádět kompletní přepis veškerých zdrojových kódů, a proto vždy uvedu pouze část řešení. Kompletní aplikace, resp. jejich zdrojové kódy budou samozřejmě k dispozici v příloze.

Další podkapitoly budou vycházet z modifikovaného členění tak, jak ho definoval při svém srovnání [Odehnal, 2011], a které se pro účel této práce hodí:

- Pravidla a doporučení pro vývoj aplikací
- Instalace a zprovoznění
- Adresářová struktura
- Nastavení aplikace
- Implementace modelu
- Implementace controllerů/presenterů
- Implementace view
- Implementace ostatních tříd
- Implementace testů

5.1 Instalace a zprovoznění

Před samotným vývojem aplikace je nutné si stáhnout běhové prostředí jazyka a knihovny frameworku a ty pak nakonfigurovat pro společný běh na webovém serveru. Popis způsobu instalace a konfigurace běhového prostředí (webserveru) nebudu detailně popisovat, protože je dalece nad rámec srovnání platforem.

5.1.1 Framework Nette

Jazyk PHP bývá dodáván společně s webovým serverem ve formě balíků (XAMPP[53], WAMP) či ho lze nainstalovat v libovolné verzi [54]jako modul do webového serveru (např.

[53]Dostupný zde http://www.apachefriends.org/en/xampp.html
[54]Dostupný zde http://php.net/downloads.php

Apache[55]) ručně. Framework Nette lze v poslední verzi stáhnout z oficiálních stránek[56], Githubu[57] či automatizovaně jako závislost pomocí nástroje Composer z repozitáře Packigist. Zde nastíním postup, který následuje po stažení frameworku z webových stránek. Samotné zprovoznění je velmi jednoduché a nevyžaduje při správném nastavení PHP a Apache žádné konfigurační zásahy. V případě, že vznikne potřeba otestovat, zda bude Nette na daném serveru fungovat, je možné spustit nástroj requirements checker, který je součástí balíku a umožňuje automatizovaně zkontrolovat, zda jsou splněny všechny nutné podmínky pro fungování frameworku v daném prostředí. Další možností je tuto kontrolu provést ručně dle požadavků na oficiálních stránkách[58]. V této práci budu využívat aktuální verzi 2.0.8.

Po stažení frameworku je v balíku tzv. sandbox, tedy adresář, který obsahuje připravenou základní kostru aplikace, což je velmi užitečné. Dále jsou přítomny pomocné nástroje (kontrola kódu, knihovna na testování Tester, ftp deployment nástroj a další), dokumentace a příklady. Postačí tedy pouze zkopírovat obsah adresáře sandbox do složky požadované složky (v závislosti na konfiguraci a webovém serveru) a po spuštění aplikace, pokud je vše v pořádku, zobrazí se uvítací stránka Nette.

5.1.2 Framework Grails

V případě Grails je situace podstatně jednodušší. Stačí pouze stáhnout celý balík[59], který obsahuje webový kontajner, zdrojové soubory frameworku, dokumentaci a pluginy. S tímto setem lze prakticky okamžitě začít vytvářet webové aplikace bez nutnosti instalovat cokoliv dalšího. V této práci budu využívat verzi 2.2.0. Minimálními požadavky jsou pouze přítomnost JDK ve verzi 1.6 a novější a samozřejmě také webový Java EE kontajner, který tuto verzi a Servlety ve verzi 2.5 a vyšší podporuje.

Grails obsahují velmi šikovnou konzolovou aplikaci, která umožňuje provádět celou řadu činností pouze napsáním jednoho příkazu. V praxi se využívá buď přímého zapisování příkazů do příkazové řádky, nebo lze využít IDE, pokud takovou funkcí disponuje, a nechat ho tyto příkazy volat automaticky pouhým stisknutím tlačítka. Záleží na preferencích konkrétního vývojáře.

[55]Dostupný zde http://httpd.apache.org/
[56] Dostupný zde http://nette.org/cs/download
[57] Dostupný zde https://github.com/nette/nette
[58] K nahlédnutí zde http://doc.nette.org/cs/requirements
[59] Dostupný zde http://www.grails.org/download

Ilustrativně níže uvádím příkaz, který lze použít pro prvotní vygenerování kostry aplikace.

```
grails create-app grailsgtd
```

5.1.3 Srovnání

Z popisu lze vidět, že Grails je v ohledu počáteční konfigurace a instalace velice propracovaný. Prakticky okamžitě lze začít tvořit samotnou aplikaci a navíc je zde k dispozici velmi dobrý konzolový nástroj, který dokáže ulehčit vývojáři život. Nette samotné je díky sandboxu také připravené k použití, ale pouze za předpokladu, že máme zprovozněný webový server. Zároveň nedisponuje žádným generátorem kódu, což je, dle roadmapy, v plánu. [Grudl, 2012]

5.2 Adresářová struktura a umístění důležitých souborů

Každý z frameworků disponuje odlišnou adresářovou strukturou. Rozhodně je tedy více než vhodné popsat, kde se jaká část nachází. Z důvodu implementace podobného architektonického vzoru lze však očekávat, že bude podobná. Celou strukturu hierarchicky zobrazím spolu s nejvýznamnějšími soubory.

5.2.1 Framework Nette

Pokud budeme uvažovat strukturu Nette v podobě jak je definována sandboxem, lze ji popsat velice jednoduše. O sandboxu se zmiňuji cíleně z toho důvodu, že Nette nepředepisuje rigidní strukturu adresářů, kterou je nutno zachovat, nýbrž si lze vytvořit v podstatě jakoukoliv dle aktuálních potřeb. I přes to však většina vývojářů využívá sandboxu s největší pravděpodobností proto, že je to nejjednodušší a z mého pohledu i nejpřehlednější varianta, jak aplikaci členit.

Celá adresářová struktura je rozdělena na složku app, která obsahuje samotnou aplikaci, resp. její logiku, libs, které slouží jako centrální úložiště použitých knihoven a rovněž obsahuje zdrojové kódy frameworku, log, do kterého se, resp. do souborů v něm obsažených, zapisují výstupy z logovacích funkcí, dále pak temp, ve kterém se nachází dočasné soubory typu cache a sessions, test obsahující případné zdrojové kódy k testům a složka www, která slouží jako vstupní bod aplikace a obsahuje, na rozdíl od ostatních složek, veřejně přímo viditelné soubory. Detailněji zpracovaný přehled struktury uvádím níže.

- App - *soubory aplikace*
    - Config - *konfigurační soubory*
        - config.neon - *hlavní konfigurační soubor*
    - model - *modely*
    - presenter - *presentery*
    - router - *routery*
    - templates - *šablony*
        - Presenter - *pohledy pro určitý presenter*
            - view.ctp - *pohled*
        - @layout.ctp - *layout*
    - boostrap.php - *úvodní/zaváděcí soubor aplikace*
- libs - *knihovny*
    - Nette - *zdrojové soubory Nette*
- log - *výstupy logovacích funkcí*
- temp - *adresář pro dočasné soubory*
    - cache - *soubory s cache*
    - cessions - *soubory se sessions*
- test - *testy pro aplikace*
- www - *složka obsahující zdroje*
    - adminer - *jednoduchý databázový správce*
    - images - *obrázky*
    - css - *kaskádové styly*
    - js - *javascript*
    - index.php - *vstupní soubor celé aplikace*

5.2.2 Framework Grails

Struktura frameworku vychází z podoby vygenerované konzolovou aplikací. Opět se nejedná o rigidní strukturu, nýbrž o výchozí. Zdrojové kódy samotné aplikace se nacházejí v adresáři grails-app, který je spolu s ostatními adresáři veřejně nepřístupný mimo složky web-app, která obsahuje přímé zdroje aplikace, jako jsou obrázky, styly a další.

- grails-app - *zdrojové kódy aplikace*
    - conf - *konfigurace*
        - BootStrap.groovy - *zaváděcí soubor*

        - DataSource.groovy - *nastavení datových zdrojů*
        - UrlMappings.groovy - *nastavení routování*
    - controllers - *controllery*
    - domain - *doménové modely*
    - i18n - *lokalizace*
    - migrations - *výstupy migračních nás. (gorm-dif)*
    - services - *soubory služeb*
    - taglib - *knihovny s tagy*
    - utils - *různé nástroje pro Grails*
    - views - *pohledy a šablony*
- lib - *knihovny*
- scripts - *gant skripty*
- src - *pomocné zdrojové soubory*
    - groovy - *soubory v jazyce groovy*
    - java - *soubory v jazyce java*
- test - *adresář se všemi testy*
    - integration - *integrační testy*
    - unit - *jednotkové testy*
- web-app - *adresář se zdroji aplikace, veřejný*
    - css - *kaskádové styly*
    - images - *obrázky*
    - js - *javascriptové knihovny*

5.2.3 Srovnání

Z adresářových výpisů je vidět, že struktura je velmi podobná. To je dle mého názoru i jedna z výhod využívání architektonických návrhových vzorů. Nejen že implementace takového vzoru vyřeší daný problém obecně uznávaným neproprietárním postupem, ale poskytuje i vývojářům, kteří znají tyto vzory, možnost rychle se zorientovat.

Pravdou je, že Grailsy mají o něco složitější strukturu. Nepřiřazoval bych však tomuto faktu žádný význam, obě struktury jsou velmi dobře čitelné.

5.3 Nastavení aplikace

V této kapitole popíšu kroky, které je nutné učinit pro počáteční nastavení aplikace. V obou frameworcích se bude jednat hlavně o nastavení připojení k databázi.

5.3.1 Framework Nette

V Nette se konfigurace prostředí provádí prostřednictvím souboru config.neon. Přípona označuje typ souboru, kterým je NEON[60]. Jedná se o speciální syntaxi velice podobnou YAMLu, což je formát pro serializaci dat. Výhodou tohoto formátu je, že není nutné zapisovat konfigurace v XML, JSONU nebo podobném výřečnějším formátu. Konfigurace je načtena po zpracování procesorem (parserem) z běžného textu, avšak zapsaného dle jasných konvencí, které spočívají ve striktním užívaní buďto pouze mezer, nebo pouze tabulátorů pro odlišení jednotlivých úrovní. V tomto souboru lze také využívat možnosti definovat několik konfiguračních sad (produkční prostředí, vývojové…). Stěžejní pro vývojáře je však sekce factories (továrny) a services (služby), kde si vydefinuje některé části, které bude napříč aplikací používat. Možnosti nastavení Nette jsou velmi bohaté, naneštěstí zde však není prostor pro kompletní popis všech prvků, které umožňují například definovat cache, chování session, parametry a podobně (pro potřeby této aplikace je postačující výchozí nastavení). Princip fungování je takový, že na základě tohoto souboru dojde po prvním překladu k vygenerování příkazů, které nastaví aplikaci a vytvoří se dependency injection kontajner.

Zdrojový kód 1 – Nette - Konfigurační soubor config,neon. Zdroj: Autor

```
common:
    parameters: # parametry dostupné napříč celou aplikací

    php: # nastavení direktiv php
        date.timezone: Europe/Prague # časová zóna

    nette: #konfigurace Nette
        application: #nastavení aplikace, debugger, catchExceptions…
            errorPresenter: Error #výchozí presenter v případě chyby

        database: # konfigurace databáze
            dsn: 'mysql:host=localhost;dbname=nettegtd'
            user: 'root'
            password:

        session: # konfigurace session, expirace, star
            expiration: 14 days
```

[60] Více informací o neonu lze získat na oficiálních stránkách zde: http://ne-on.org/

```
    services: # generátor služeb
        authenticator: Authenticator
        routerFactory: RouterFactory
        router: @routerFactory::createRouter
        taskModel: Models\Task
        tagModel: Models\Tag
        userModel: Models\User
        stateModel: Models\State
        taskTagModel: Models\TaskTag
        commentModel: Models\Comment

    factories: # generátor továren
```

Další možností, jak ovlivnit nastavení aplikace je bootstrap.php, neboli zavaděč. V tomto souboru dochází k načtení knihoven, konfiguračních souborů, nastavení složek pro dočasné soubory, nastavení debug módu a samozřejmě inicializaci RobotLoaderu, což je třída, která se stará o načítání používaných tříd bez nutnosti je explicitně pokaždé definovat. Kouzlo RobotLoaderu tkví v tom, že dokáže identifikovat závislosti mezi třídami navzájem a rozhraními a načte je ve správném pořadí.

Zdrojový kód 2 – Nette - Zaváděcí soubor bootstrap.php. Zdroj: Autor

```php
<?php

// Načtení Nette frameworku
require __DIR__ . '/../libs/autoload.php';

// Nová instance třídy configurator (konfigurace, DI, ...)
$configurator = new Nette\Config\Configurator;

$configurator->setDebugMode(FALSE); // zapnutí Nette Debuggeru (Laděnky)

// umístění logovacích souborů
$configurator->enableDebugger(__DIR__ . '/../log');

// definování složky pro dočasné soubory
$configurator->setTempDirectory(__DIR__ . '/../temp');
// RobotLoader - načtení všech potřebných souborů
$configurator->createRobotLoader()
    ->addDirectory(__DIR__)
    ->addDirectory(__DIR__ . '/../libs')
    ->register();

// vytvoření DI kontajneru z config.neon
$configurator->addConfig(__DIR__ . '/config/config.neon');
$container = $configurator->createContainer();

return $container;
```

Poslední částí, je možnost definovat vlastní routování, tedy převod adresy na konkrétní akce presenteru. Tuto funkcionalitu ale nevyužiji, protože si vystačím s výchozím nastavením,

které mapuje dle vzoru /controller/action/id a je naprosto dostačující. Jedinou věcí byla pouze změna domovské stránky na Task:default, tedy na controller task a akci default.

Zdrojový kód 3 – Nette – Nastavení routování v souboru RouterFactory.php. Zdroj: Autor

```
class RouterFactory
{
    public function createRouter()
    {
        $router = new RouteList();
        $router[] = new Route('index.php', 'Task:default',
                              Route::ONE_WAY);
        $router[] = new Route('<presenter>/<action>[/<id>]',
                             'Task:default');
        return $router;
    }
}
```

5.3.2 Framework Grails

Grails mají konfiguraci rozdělenu do více souborů, resp. tříd či closures, z nichž se všechny nachází ve složce conf.

V Bootstrap.groovy lze zasáhnout do inicializačního procesu a nechat si například předvyplnit databázi testovacími daty.

Zdrojový kód 4 – Zaváděcí soubor bootstrap.groovy frameworku Grails. Zdroj: Autor

```
class BootStrap {
    def init = { servletContext ->
    }
    def destroy = {

    }
}
```

UrlMappings.groovy slouží pro nastavení routování, resp. mapování adres na jednotlivé controllery a jejich akce. V konfiguraci routování bylo potřeba také upravit pouze domovskou stránku na controller task a akci list.

Zdrojový kód 5 – Grails - Nastavení routování UrlMappings.groovy. Zdroj: Autor

```
class UrlMappings {
    static mappings = {
        "/$controller/$action?/$id?"{}
        "/"(controller: "task", action: "list")
        "500"(view:'/error')
    }
}
```

DataSource.groovy je konfigurační soubor persistentní vrstvy pro přístup k požadovanému databázovému systému, kde lze i poměrně jednoduše nakonfigurovat objektově relační mapper GORM (Hibernate).

Zdrojový kód 6 – Grails – Nastavení datových zdrojů DataSource.groovy. Zdroj: Autor

```
dataSource {
    pooled = true
    dbCreate = "create-drop"
    url = "jdbc:mysql://localhost/grailsgtd?autoReconnect=true&
           useUnicode=yes&characterEncoding=UTF-8"
    driverClassName = "com.mysql.jdbc.Driver"
    username = "root"
    password = ""
    loggingSql = true
}
```

O odstavec výše jsem představil nejpoužívanější konfigurační soubory. Mezi méně často upravované, ale o to komplexnější, patří Config.groovy, který slouží pro konfiguraci detailů fungování frameworku, BuildConfig.groovy, ve kterém lze nakonfigurovat závislosti aplikace, ApplicationResources.groovy, který umožňuje definovat centrální místo pro rozdělení zdrojových souborů (javascript, styly a další) do modulů a tyto moduly pak jednoduše načítat v layoutech. Posledním konfiguračním souborem je resources.groovy, který slouží jako konfigurační soubor pro vydefinování jednotlivých tříd vytvářených a injektovaných dependency injection kontajnerem.

5.3.3 Srovnání

Komplexnost základních konfiguračních souborů je u obou frameworků shodná, jednotlivé konfigurační direktivy jsou přehledně členěny. Z mého pohledu musím však podotknout, že konfigurace pokročilejších funkcionalit v Nette se mi zdála bez jakýchkoliv znalostí mírně neintuitivní (config.neon), naproti tomu Grails jsem dokázal zprovoznit prakticky bez dokumentace. Tato skutečnost však byla možná způsobena složitějším přístupem k implementaci modelové část v Nette.

5.4 Implementace modelu

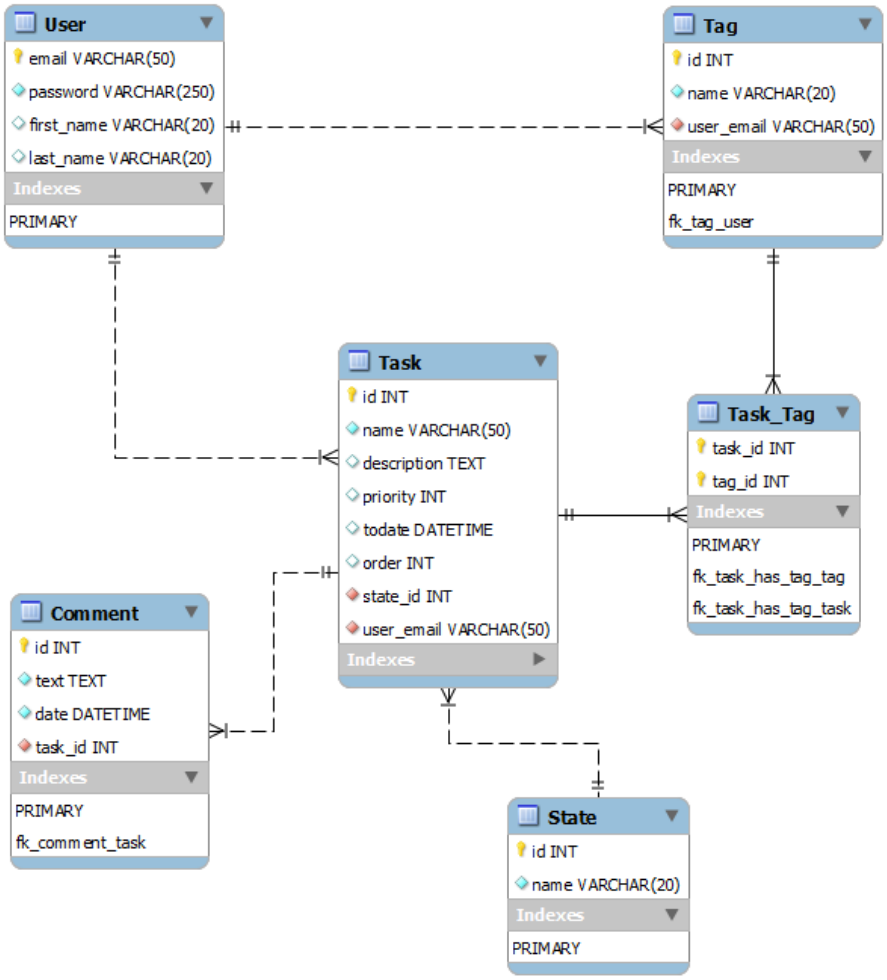
Při implementaci modelu lze postupovat různými způsoby. Předpokládejme však, že v obou frameworcích budou využity striktně možnosti, které jsou obsaženy již v základu. V Nette budu postupovat metodou database-first, kdy nejdříve vytvořím databázové tabulky a poté na jejich základě modely. V Grails zvolím opačný postup, tedy code-first, nejdříve

naprogramuji modely a na jejich základě nechám framework vygenerovat databázové tabulky. V Nette nelze v základu code-first přístup využít, protože tuto funkcionalitu zkrátka nemá, ale zdá se mi zajímavé pro ilustraci rozdílů tuto techniku využít (na práci s těmito modely nemá volba přístupu k tvorbě zásadní vliv).

5.4.1 Framework Nette

Framework Nette nabízí v základu poměrně dobrý nástroj pro jednoduchou práci s databází Nette/Database, který vychází z výborné knihovny NotORM. Napříč celou knihovnou je aplikován přístup líného načítání. Pokud by tedy nedošlo k využití získaných dat z databáze, knihovna tento dotaz ani neprovede. Samozřejmostí jsou pak i další optimalizace, které podstatně zrychlují běh celé aplikace.

Níže uvádím databázové schéma, které jsem vytvořil pro potřeby této aplikace. Jedná se o běžné schéma pro entitně-relační databázové systémy.



Obrázek 18 – Nette - Entitně-relační diagram databáze aplikace. Zdroj: Autor

Přístupů, jak implementovat doménový model, může být celá řada. Abych dodržel čistý postup a demonstroval možnosti frameworku, definuji si nejdříve společného abstraktního předka pro modely, který poskytne základní funkcionalitu pro práci s databází a oprostí programátora od neustále se opakujícího kódu. Tento předek je k dispozici na oficiálních stránkách[61], já jsem jej však upravil pro práci s Nette/Database a nikoliv přímo s NotORM.

Dále pak naimplementuji postupně všechny modely. Na rozdíl od Grails, NotORM, resp. Nette/Database nepracuje s vazbou M:N a je třeba ji pojmout jako 1:N a N:1. Nicméně komfort práce je i tak poměrně dobrý. Pokud by tento „problém“ programátor chtěl řešit elegantněji, využil by plnohodnotný entitně relační mapper, například Doctrine[62].

V aplikaci jsem se rozhodl využívat dependency injection a tudíž je nutné modifikovat i konfigurační soubor config.neon, viz aplikace v příloze. Pro lepší kontrast s Grails zde uvádím zdrojový kód modelu User. Třída je mimo dvou pomocných metod prázdná a atributy zde nemají žádný význam. Pomocí modelu je zde realizován přístup do konkrétní databázové tabulky.

Zdrojový kód 7 – Nette – Model User. Zdroj: Autor

```
class User extends Base
{
    public function validateLogin($email, $password)
    {
        $user = $this->getConnection()->where(array(
            'email' => $email,
            'password' => self::calculateHash($password)
        ))->fetch();

        return $user;
    }

    public static function calculateHash($password)
    {
        if ($password === Strings::upper($password)) {
            $password = Strings::lower($password);
        }
        return crypt($password, '$2a$07$');
    }

}
```

[61] http://pla.nette.org/cs/nacitani-modelu-s-notorm-a-dependency-injection
[62] Více informací je k dispozici zde: http://www.doctrine-project.org/

5.4.2 Framework Grails

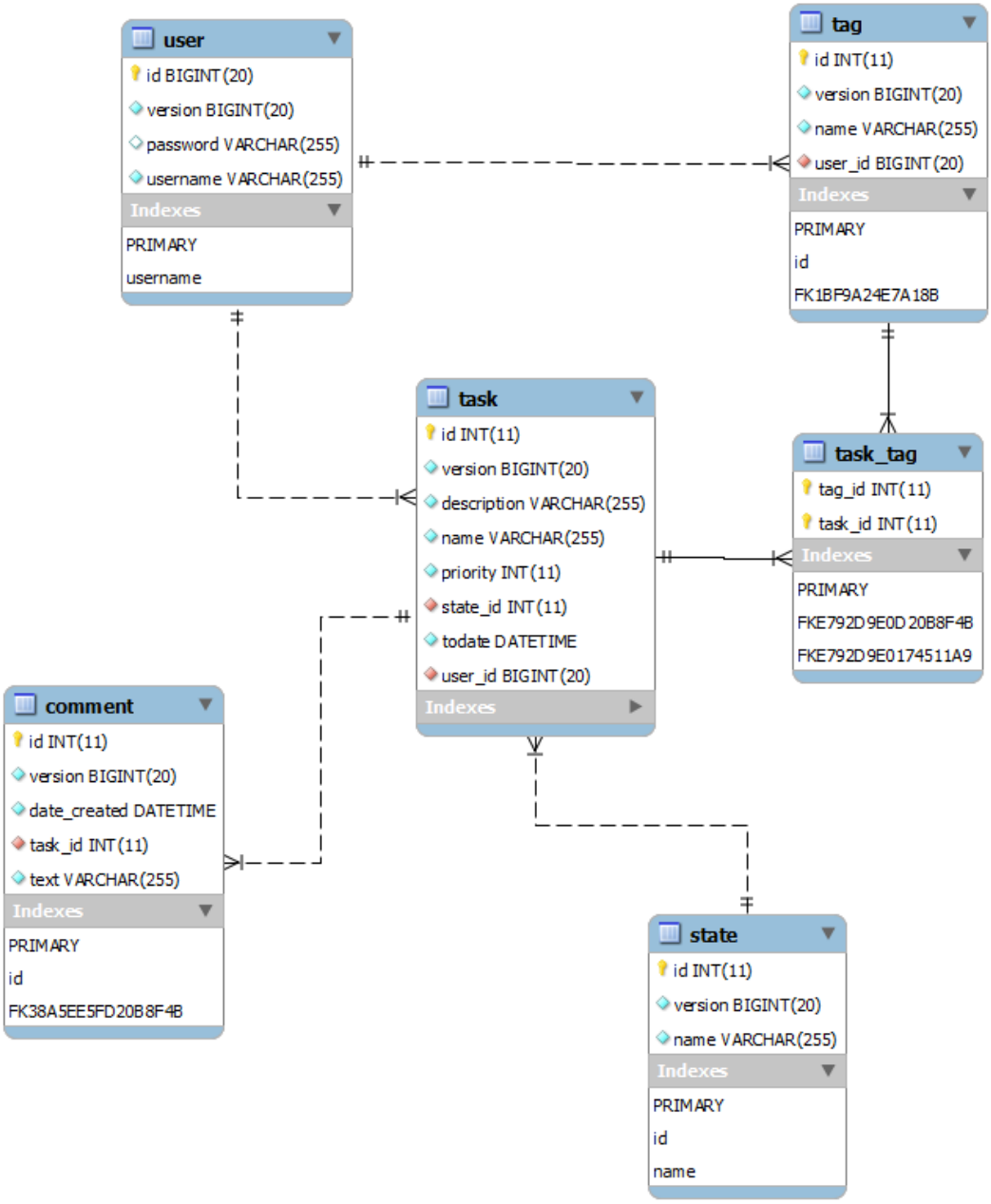
Jak jsem již zmínil, v tomto frameworku jsem zvolil metodu code-first a nechal jsem si tedy vygenerovat databázové tabulky automatizovaně. Generování probíhá vždy při správném nastavení hodnoty dbCreate (například na drop-create, tedy smazání existujícího a vytvoření nového aktuálního schématu) v nastavení databázové persistentní vrstvy DataSource.config.

Samotná tvorba modelové třídy je jednoduchá a spočívá v nadefinování atributů formou properties. Dále lze nastavit constraints (omezení hodnot a validace) a nepersistentní nebo jinak také transientní atributy prostřednictvím pole transients. Díky tomu, že jádrem GORMu je Hibernate, lze využít i události (například níže definovaný beforeInsert, který v tomto případě před prvotním vložením do databáze provede heshování hesla). Znovu nadefinovaná a přepsaná metoda toString je zde z toho důvodu, že v případě výpisu objektu User (třeba v rozbalovacím seznamu) se zobrazí uživatelské jméno.

Zdrojový kód 8 – Grails – Model User. Zdroj: Autor

```
class User {
    /* Parametry */
    String username
    String password

    /* Transientní parametry pro registraci */
    String passwordRegistration
    String passwordConfirm

    static transients = ['passwordRegistration','passwordConfirm']

    @Override
    def String toString(){
        return username
    }

    /* Omezení a validace */
    static constraints = {
        username unique: true, blank: false, email: true
        password nullable: true
        passwordConfirm bindable:true
        passwordRegistration bindable:true, blank:false, validator:{ v
          al, obj ->
            if(obj.passwordRegistration != obj.passwordConfirm)
                return 'user.passwords.notequal'
        }
    }

    /* Hashování hesla */
    def beforeInsert(){
        password = passwordRegistration.encodeAsMD5()
    }
}
```

Poté, co dojde ke kompilaci a následnému publikování aplikace do webového kontajneru, aplikace vygeneruje databázi na základě definovaných tříd. Ke standardním atributům přidává ve výchozím nastavení GORM další atribut version, který slouží jako řešení konkurenčnosti namísto zamykání. Obrázek 19 zachycuje vygenerované databázové schéma.



Obrázek 19 – Grails - Entitně-relační diagram databáze aplikace. Zdroj: Autor

5.4.3 Srovnání

Srovnání obou frameworků je, co se týče modelu, velmi obtížné a každý jde jinou cestou. Grails obsahuje poměrně komplexní persistentní vrstvu, naproti tomu podpora

databáze v Nette je velmi jednoduchá a troufám si říci, že účinná. V případě, že by v Nette základní nástroj, tedy Nette/Database nestačil, lze využít nějakou z robustnějších knihoven implementujících některý z pokročilejších návrhových vzorů (UnitOfWork, DataMapper apod.).

Na relačních schématech je vidět, že databáze jsou v obou případech v podstatě identické a přístup code-first funguje velmi dobře.

Rozdíl mezi frameworky je nejvíce patrný při vykonávání příkazů z controlleru, kdy se pracuje s oběma vrstvami ve frameworcích velmi odlišně. Řešení plnohodnotným ORM se jeví jako velmi dobrý přístup, naneštěstí jsem se poměrně často setkával při vývoji s chybovými hlášeními, které byly v podstatě nic neříkající a trvalo delší chvíli, než se mi je podařilo vyřešit.

5.5 Implementace controllerů/presenterů

Na základě návrhu či definovaných modelů lze začít s tvorbou controllerů v případě Grails a presenterů v případě Nette, které jsou zodpovědné za aplikační logiku. Podobně jako v předchozích částech oba frameworky k dané problematice přistupují odlišně. Základní charakteristiky se pokusím nastínit v následujících podkapitolách.

5.5.1 Framework Nette

Jako ilustraci tvorby presenteru použiji UserPresenter.php, který se stará o registraci a přihlašování/odhlašování uživatelů. V Nette by měl každý presenter přímo či nepřímo dědit od třídy BasePresenter, která dědí od třídy Presenter. Význam spočívá především v možnosti nadeklarovat události společné všem presenterům v jednom místě. Analogicky lze tento fakt využít i pro funkcionalitu společnou více třídám (pravidlo DRY). Třída Presenter neposkytuje pouze základní metody pro práci s požadavky a odpověďmi, nýbrž i prakticky s celým frameworkem.

Díky tomu, že Nette disponuje dependency injection kontajnerem, je možné si nechat nainjektovat závislosti v podobě modelů automatizovaně, například pomocí setter injection (aplikace DI kdy dochází k injektáži závislostí přes určenou metodu, další variantou může být constructor). Při deklaraci parametrů setter metody je nasnadě neopomenout definovat i type

hint[63], díky kterému kontajner pozná, jakou závislost, resp. třídu má injektovat. V případě, že chceme využívat automatického injektování, je nutné definovat v presenteru metodu, která musí být ve tvaru inject… (injectModels, injectMail,…). Tak bude Nette vědět, že má volat tyto metody ihned po vytvoření instance automatizovaně a díky type hintu do nich vloží potřebné závislosti. Takové řešení je nejen přehledné, funkční a čisté, ale i velice jednoduché. Díky použití dependency injection není nutné pro tyto účely využívat context aplikace.

Zdrojový kód 9 – Nette – Presenter User – injektáž závislostí. Zdroj: Autor

```
protected $userModel;

public function injectModels(\Models\User $userModel)
{
    if ($this->userModel) {
        throw new Nette\InvalidStateException('Injektáž již proběhla.');
    }

    $this->userModel = $userModel;
}
```

Nette klade velký důraz na znovupoužitelnost kódu a tak zavedlo a plně podporuje komponentový přístup (formuláře, menu, ankety, komentáře…). Tyto komponenty se vkládají do pohledů prostřednictvím továren[64] v presenterech, které vytvoří instanci dané komponenty a vrátí ji tomu, kdo si ji vyžádal. Na oficiálních stránkách v sekci doplňky můžeme najít mnoho hotových komponent, které vytvořili uživatelé či tvůrci frameworku. Komponentu lze chápat jako relativně soběstačný prvek, který má svoji aplikační i prezentační logiku. Další částí User presenteru tedy bude továrna (české pojmenování návrhového vzoru Factory) na registrační formulář a jeho obsluha. Na formuláři je dobře vidět, jak je poskládán z jednotlivých dalších komponent ve formě políček a tlačítek. Na jednotlivá políčka jsou aplikována validační pravidla. Tato pravidla, v případě, že jsou ze sady předpřipravených validátorů, mají pak ve view automaticky vygenerován javascriptový validátor (ve výchozím způsobu generování formulářů). Je tedy automaticky k dispozici jak validace na straně klienta, tak na straně serveru. Na konci formuláře je tlačítko odeslat připojena obslužná metoda regFormSucceeded s parametrem, ve kterém je odeslaný formulář. Jedná se v podstatě o callback. V obsluze pak v případě správného vyplnění dojde k vygenerování hesla, uložení uživatele a přesměrování na hlavní stranu.

[63] Type hint je identifikátor typu dané proměnné, omezuje tedy množinu vkládaných hodnot na hodnoty jím specifikované. V praxi se často jedná buď o konkrétní název třídy či rozhraní.
[64] Návrhový vzor factory

Zdrojový kód 10 – Nette – Presenter User – registrační formulář. Zdroj: Autor

```
protected function createComponentRegForm()
{
    $form = new UI\Form;
    $form->addText('email', 'E-Mail:')
        ->setRequired('Vyplňte váš email.')
        ->addRule(Form::EMAIL, 'Neplatný typ e-mailu.')
        ->addRule(function (\Nette\Forms\IControl $control) {
            return ($this->userModel->fetchAll()->where('email', $control-
>getValue())->count() > 0) ? FALSE : TRUE;
        }, 'Toto uživatelské jméno již existuje.');

    $form->addPassword('password', 'Heslo:')
        ->setRequired('Vyplňte vaše heslo.');

    $form->addPassword('password1', 'Heslo:')
        ->setRequired('Vyplňte heslo pro kontrolu.')
        ->addRule(Form::EQUAL, 'Hesla se neshodují', $form['password']);


    $form->addSubmit('send', 'Registrovat');

    $form->onSuccess[] = $this->regFormSucceeded;
    return $form;
}


public function regFormSucceeded($form)
{
    if ($form->isSubmitted()) {
        $values = $form->getValues();

        unset($values['password1']);
        $values['password'] = $this->userModel-
>calculateHash($values['password']);

        if ($this->userModel->insert($values)) {
            $this->flashMessage('Registrace proběhla úspěšně, můžete se
přihlásit.', 'success');
            $this->redirect('in');
        } else {
            $this->flashMessage('Registrace se nezdařila, tento email
zřejmě v databázi již existuje.', 'error');
        }
    }
}
```

Další částí presenteru je přihlašování a odhlašování uživatelů. Továrnu na formulář zde již nemá význam uvádět, proto uvedu pouze obsluhu formuláře. Autentizaci a případně i autorizaci (kterou jsem řešil na úrovni BasePresenteru prostou kontrolou, zda je uživatel přihlášen) uživatelů obstarává služba frameworku Nette/Security/User. Pro tento účel je potřeba mít definovaný model implementující rozhraní IAuthenticator předepisující metodu authenticate, která na základě parametru ověří autenticitu a v případě správnosti vrátí vytvořený objekt Nette/Security/User, který je uložen do session a dostupný celou dobu

platnosti přihlášení, například pomocí metody presenteru getUser(). Odhlašování probíhá pak jednoduše zavoláním metody logout na instanci tohoto objektu.

Zdrojový kód 11 – Nette – UserPresenter – přihlašování a odhlašování uživatelů. Zdroj: Autor

```
public function signInFormSucceeded($form)
{
    $values = $form->getValues();

    if ($values->remember) {
        $this->getUser()->setExpiration('+ 14 days', FALSE);
    } else {
        $this->getUser()->setExpiration('+ 20 minutes', TRUE);
    }

    try {
        $this->getUser()->login($values->email, $values->password);
    } catch (Nette\Security\AuthenticationException $e) {
        $this->flashMessage('Přihlášení se nezdařilo, bylo zadáno neplatné
jméno nebo heslo.', 'error');
        return;
    }

    $this->redirect('Task:default');
}

public function actionOut()
{
    $this->getUser()->logout();
    $this->flashMessage('Byl/a jste úspěšně odhlášen/a.', 'success');
    $this->redirect('in');
}
```

V případě, že není nutná pro dané view žádná obsluha, je možné metodu render či action k danému pohledu vynechat a framework jen při volání patřičné adresy zobrazí daný pohled. Příkladem jsou metody in (pro přihlášení) a metoda reg(pro registraci). V kódu nejsou nijak zastoupeny a slouží pouze pro zobrazení komponent ve formě formulářů.

5.5.2 Framework Grails

V této části popíšu tvorbu controlleru ve frameworku Grails, kde díky tomu, že je k dispozici sofistikovaný generátor, je situace podstatně jednodušší. Generátor dokáže na základě vytvořeného modelu vygenerovat patřičný controller, který bude s tímto modelem pracovat. Dá se říci, že po vygenerování je třída plně provozuschopná a má předdefinovány modifikované základní CRUD operace. Po spuštění aplikace vše funguje. Díky tomu, že je však tato třída omezena pouze na registraci a přihlášení/odhlášení, bylo potřeba většinu metod, resp. akcí, smazat. Zůstala jen metoda pro vytváření uživatelů. Nově bylo potřeba vytvořit akce pro přihlášení a odhlášení. Zpravidla se objevují vždy metody dvě, jedna pro zobrazení daného formuláře a další metoda pro jeho obsluhu. Význam spočívá i v možnosti

striktně omezit HTTP metodu přístupu pro danou akci, či jen prostě provádět odlišné akce, jak v případě zobrazení formuláře, tak jeho obsluhy přehledněji, než by tomu bylo v případě, že bychom sloučili obě akce do jedné a pouze kontrolovali, zda došlo k odeslání.

Zdrojový kód 12 – Grails – UserController – povolené metody pro přístup k jednotlivým akcím. Zdroj: Autor

```
static allowedMethods = [save: "POST"]
```

Zpracování formuláře je jednoduché. Vytvoří se nový objekt dle parametrů odeslaných z formuláře. Framework provede automatický binding na odpovídající atributy daného objektu. Při pokusu o uložení tento objekt zvaliduje a dle toho, zda validace prošla, vrátí pravdivostní hodnotu.

Zdrojový kód 13 – Grails – UserController – registrace uživatelů. Zdroj: Autor

```
def registration() {
    [userInstance: new User(params)]
}

def save() {
    def userInstance = new User(params)

    if (!userInstance.save(flush: true)) {
        render(view: "registration", model: [userInstance: userInstance])
    return
    }

    flash.message = message(code: 'user.registration.message')
    redirect(controller: 'user', action: 'login')
}
```

Přihlašování je v Grails řešitelné množstvím více či méně sofistikovaných pluginů. Je možné si však vystačit s pouhým uložením přihlášeného uživatele do odpovídající session. Lze si povšimnout, že základní mechanismus práce s modelem při vyhledávání představuje poměrně velmi magické metody pro vyhledávání findBy….

Zdrojový kód 14 – Grails – UserController – Přihlašování a odhlašování uživatelů. Zdroj: Autor

```
def login = {

}

def doLogin = {
    def user = User.findByUsernameAndPassword(params['username'],
params['password'].encodeAsMD5())

    session.user = user

    if (user) {
        redirect(controller: 'task', action: 'list')
    } else {
```

```
        flash.message = message(code: 'user.login.fail')
        render(view: "login")
    }
}

def logout = {
    session.invalidate()
    flash.message = message(code: 'user.logout')
    redirect(controller: 'user', action: 'login')
}
```

5.5.3 Srovnání

Porovnávat mezi sebou controller v Grails a presenter v Nette na třídách, které řeší stejnou problematiku, je velmi zajímavé. Oba frameworky jdou naprosto odlišnou cestou. Názorně lze vidět, že Nette soustřeďuje logiku aplikace do presenteru a view využívá v podstatě pouze jako „hloupý“ zobrazovač dat. Naproti tomu Grails přenechávají prezentační vrstvě podstatně větší práci a zároveň i modelu (záleží zde hodně na přístupu, který si vývojář k implementaci domény vybere). Netroufám si tvrdit, který přístup je lepší či horší, volba bude pravděpodobně záležet na preferencích daného vývojáře. Dle mého názoru je však o něco více logicky členěný přístup používaný Grails. Pravděpodobně je to však způsobeno daným architektonickým vzorem. Významným nástrojem je zde i generátor kódu, který dokáže prakticky bez jakéhokoliv zásahu vytvořit funkční CRUD aplikaci.

Pokud bych měl srovnat práci s doménami, přístup Nette se mi zdá průhlednější, i přes to, že je nepoměrně jednodušší a s méně možnostmi, než nabízí konkurenční framework. Grails je plné magických metod a celkově se tváří velice netransparentně. Pro ilustraci bych zmínil třídu User a její magické metody. Dostupnost této třídy (a nejen této, ale všech doménových tříd) je ve všech částech controlleru. Pravděpodobně je to záměr, naneštěstí toto téma však překračuje rozsah práce a nelze se mu zde patřičně věnovat.

5.6 Implementace view

V této podkapitole se budu zabývat implementací jednoho vzorového pohledu u obou frameworků a samozřejmě také výchozím layoutem. Layout slouží jako globální šablona, ve které dochází k zobrazování konkrétních pohledů. Výhody tohoto přístupu jsou jasné, centralizovaná správa vzhledu, splnění pravidla DRY, celkově pohodlnější a rychlejší práce.

5.6.1 Framework Nette

Šablonovací systém (Latté) je podle mého názoru jednou z největších předností Nette, minimálně při porovnání s ostatními PHP frameworky. Základními stavebními kameny jsou bloky, makra a helpery. Význam bloků spočívá v možnosti definovat v šabloně místa, která mají být použita pro vkládání obsahu. Jednoduše lze tedy mapovat samotný obsah, menu a další na konkrétní místa v layoutu. Makra pak především zjednodušují práci s vypisováním escapovaných proměnných, formulářů, větvících konstrukcí a dalších. Odpadá tedy nutnost zapisovat běžné konstrukce jazykem PHP. Helpery jsou jednoduché funkce, které snadno přeformátují nebo upraví data do požadované podoby. Příkladem může být zkracování řetězců, formátování data a času a mnoho dalších. Latté tedy staví na zjednodušení, zrychlení a větší bezpečnosti (automatické výchozí escapování).

Níže uvádím zdrojový kód výchozího layoutu. Lze si všimnout, že jsou použity dva bloky, a to title a content. Title slouží pro vkládání titulku a content pro samotného obsahu. Dále jsou vidět makra n:foreach (n je jmenný prostor pro nette), n:href, které slouží pro vygenerování odkazu pomocí nastavení v routování a další.

Zdrojový kód 15 – Nette – Šablona layout.latte. Zdroj: Autor

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Nette GTD - {block title|striptags}{/block}</title>
    <link rel="stylesheet" media="screen,projection,tv"
href="{$basePath}/css/screen.css">
    <script src="{$basePath}/js/jquery.min.js"></script>
    <script src="{$basePath}/js/netteForms.js"></script>
    <script src="{$basePath}/js/netteAjax.js"></script>
</head>

<body>

<div id="Container">
    <h1>Nette GTD</h1>

    <a n:if="$user->isLoggedIn()" n:href="User:out" class="button
logout">Odhlásit</a>
    <div n:foreach="$flashes as $flash" class="alert alert-{$flash->type}
alert-block">
        <h3>Upozornění!</h3>
        {$flash->message}
    </div>

    {block #content}{/block}
</div>
</body>
</html>
```

Dále přikládám zdrojový kód šablony pro registraci uživatele. Je skutečně jednoduchý. Definuje, že obsah se bude zobrazovat v bloku content a nadpis se bude automaticky pomocí vnořeného makra zobrazovat i v bloku title. Dalším prvkem je pak už pouze makro control, která automatizovaně vypíše prvky formuláře tak, jak byly nadefinovány v továrně v presenteru. Lze využít i možnosti celý formulář definovat ručně, což se vyplatí zejména u nestandardních formulářů, kde výchozí renderer/vykreslovač nedělá přesně to, co očekáváme.

Zdrojový kód 16 – Nette – šablona user/reg.ctp. Zdroj: Autor

```
{block #content}

<h2 n:block="title">Registrace</h2>

{control regForm}
```

5.6.2 Framework Grails

V Grails je situace opět o něco jednodušší. Konzolová aplikace na generování kódu umí na základě controlleru a modelu vygenerovat odpovídající view pro základní CRUD operace. Z modelu načte všechna omezení a vazby. Na jejich základě vygeneruje formuláře, detaily dané položky či přehledové tabulky se stránkováním. Vývoj tedy nemusí začínat nutně od nuly, ale lze začít s předpřipravenou základní funkční aplikací, která se posléze upraví tak, aby odpovídala specifikaci.

Šablonovací systém se nazývá GSP (Groovy Server Pages), což je analogie k JSP (Java Server Pages). Grails používají podobný přístup jako Nette, tedy rozlišení na layouty a pohledy. V layoutu se definují bloky (head, title a body), do kterých se posléze mapuje obsah z jednotlivých pohledů. Grails zavádí svoji speciální sadu předpřipravených tagů, které mají za cíl zjednodušit často se opakující úkony elegantním způsobem a pozitivně působit na minimalizaci tzv. skriptletů[65]. Najdeme zde tagy pro jednoduchý foreach, řazení, stránkování, formuláře, inputy a další.

Zdrojový kód 17- Grails – layouts/main.gsp. Zdroj: Autor

```
<!DOCTYPE html>
<!--[if lt IE 7 ]> <html lang="en" class="no-js ie6"> <![endif]-->
<!--[if IE 7 ]>    <html lang="en" class="no-js ie7"> <![endif]-->
<!--[if IE 8 ]>    <html lang="en" class="no-js ie8"> <![endif]-->
<!--[if IE 9 ]>    <html lang="en" class="no-js ie9"> <![endif]-->
<!--[if (gt IE 9)|!(IE)]><!--> <html lang="en" class="no-js"><!--
<![endif]-->
```

[65] Skriptlet je fragment kódu zapsaný přímo v prezentační vrstvě. Obecně se snaží tomuto jevu předcházet a umisťovat logiku aplikace do presenteru/controlleru, tedy do vrstvy s aplikační logikou.

```
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title><g:layoutTitle default="GrailsGTD"/></title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="shortcut icon" href="${resource(dir: 'images', file:
'favicon.ico')}" type="image/x-icon">
    <link rel="apple-touch-icon" href="${resource(dir: 'images', file:
'apple-touch-icon.png')}">
    <link rel="apple-touch-icon" sizes="114x114" href="${resource(dir:
'images', file: 'apple-touch-icon-retina.png')}">
    <link rel="stylesheet" href="${resource(dir: 'css', file:
'main.css')}" type="text/css">
    <link rel="stylesheet" href="${resource(dir: 'css', file:
'mobile.css')}" type="text/css">
    <g:layoutHead/>
    <r:layoutResources/>
</head>

<body>
<div id="grailsLogo" role="banner">
    <a href="http://grails.org">
        <img src="${resource(dir: 'images', file: 'grails_logo.png')}"
alt="Grails"/>
    </a>
    <g:if test="${session.user}">
        <g:link controller="user" action="logout"
class="logout"><g:message code="user.logout"/></g:link>
    </g:if>
</div>
<g:layoutBody/>

<div class="footer" role="contentinfo"></div>

<g:javascript library="application"/>
<r:layoutResources/>

</body>
</html>
```

Pro srovnání dále uvádím zdrojový kód registračního formuláře. Zde je názorně vidět odlišný přístup k této problematice. Navíc stránka pro registraci neobsahuje formulářová pole přímo, ale vkládá je z další šablony (kvůli možnému znovupoužití kódu). Celé řešení je poměrně „upovídané“, ale logické, vývojář se intuitivně neztrácí.

Zdrojový kód 18 - Grails – user/registration.gsp. Zdroj: Autor

```
<!DOCTYPE html>
<html>
<head>
    <meta name="layout" content="main">
    <title><g:message code="user.registration.label"/></title>
</head>

<body>

<div class="nav" role="navigation">
    <ul>
```

```
        <li><a class="home" href="${createLink(action:
'login')}"><g:message code="user.registration.back.label"/></a></li>
    </ul>
</div>

<div id="create-user" class="content scaffold-create" role="main">

    <h1><g:message code="user.registration.label"/></h1>

    <g:if test="${flash.message}">
        <div class="message" role="status">${flash.message}</div>
    </g:if>

    <g:hasErrors bean="${userInstance}">

        <ul class="errors" role="alert">
            <g:eachError bean="${userInstance}" var="error">
                <li <g:if test="${error in
org.springframework.validation.FieldError}">data-field-
id="${error.field}"</g:if>><g:message
                        error="${error}"/></li>
            </g:eachError>
        </ul>

    </g:hasErrors>

    <g:form action="save">

        <fieldset class="form">
            <g:render template="form"/>
        </fieldset>

        <fieldset class="buttons">
            <g:submitButton name="create" class="save"
value="${message(code: 'user.registration.submit.label')}"/>
        </fieldset>

    </g:form>
</div>
</body>
</html>
```

Šablonovací systém využívá lehce odlišný, méně úsporný přístup a je nutné neustále opakovat i značky jako je head, body, html a další. Zřejmě nemá význam popisovat jednotlivé značky v kódu, jedná se v podstatě jen o zpracování proměnných předaných do view a zobrazení samotného formuláře včetně validačních chyb. Jednou ze zajímavostí je i využívání lokalizací pomocí slovníkového přístupu (klíč/hodnota pro každý jazyk). K tomuto bodu se ještě vrátím v podkapitole 5.7. Další částí je tedy fragment kódu s políčky registračního formuláře.

Zdrojový kód 19 - Grails – user/_form.gsp. Zdroj: Autor

```
<div class="fieldcontain ${hasErrors(bean: userInstance, field:
'username', 'error')} required">
    <label for="username">
        <g:message code="user.username.label"/>
        <span class="required-indicator">*</span>
    </label>
    <g:field type="email" name="username" required=""
value="${userInstance?.username}"/>
</div>

<div class="fieldcontain ${hasErrors(bean: userInstance, field:
'passwordRegistration', 'error')} required">
    <label for="passwordRegistration">
        <g:message code="user.password.label"/>
        <span class="required-indicator">*</span>
    </label>
    <g:passwordField name="passwordRegistration"/>
</div>

<div class="fieldcontain ${hasErrors(bean: userInstance, field:
'passwordConfirm', 'error')} required">
    <label for="passwordConfirm">
        <g:message code="user.password.label"/>
        <span class="required-indicator">*</span>
    </label>
    <g:passwordField name="passwordConfirm"/>
</div>
```

Opět se jedná o poměrně jednoduchý mechanismus, kdy pomocí tagů message vypíše GSP lokalizovaný nadpis políčka. Dále pak samotné políčko a jako počáteční hodnotu nastaví dle předané proměnné userInstance. Podobně jako v Nette dochází ve výchozím stavu při každém výpisu proměnné k automatickému escapování. Tagy lze používat nejen zabudované, ale i již vytvořené od ostatních vývojářů, či je vytvářet osobně.

5.6.3 Srovnání

Jak jsem již naznačil, prezentační vrstva je silnou stránkou Nette. Na první pohled je možné vidět rozdíl mezi oběma frameworky. Nette vede v jednoduchosti, jeho view jsou velmi úsporná a přehledná. Pozitivně však hodnotím přítomnost vlastních značek u Grails. Jedná se o podobný přístup, jako má Nette. Nemohu se však ubránit pocitu určité těžkopádnosti (obzvlášť v kontrastu s Latté).

5.7 Implementace ostatních tříd

Tato podkapitola se zaměřuje na implementaci dalších částí aplikace, které nespadají ani do jedné z předchozích částí.

5.7.1 Framework Nette

V Nette není potřeba nastavovat či implementovat žádné dodatečné třídy.

5.7.2 Framework Grails

Jak jsem se již zmínil v kapitole 5.6.2, je potřeba implementovat překlady pomocí tabulky, či jinak řečeno podle slovníku. Grails využívají lokalizace statických prvků na bázi kolekce klíč/hodnota, kdy pro každou lokalizaci existuje jeden soubor. V tomto souboru se nacházejí nejen překlady z pohledů, ale i z řadičů a modelů. Všechny lokalizace jsou tedy na jediném místě. Identifikace jednotlivých překladů je založena na tečkové navigaci, což je v podstatě něco jako pole. Je samozřejmé, že používat slovníkové lokalizace není povinnost, ale u složitějších aplikací, kdy se počítá s tím, že by mohla vzniknout v budoucnu potřeba lokalizace, je to nanejvýš vhodné. Lokalizace se nacházejí ve složce i18n v aplikační části, kde je každý soubor identifikován konvenčně tvarem messages_cs_cz.properties. Složenina cs_cz je identifikátor jazyka[66] a země[67]. Níže uvádím několik příkladů.

Zdrojový kód 20 – Grails – lokalizační soubor message_cs_cz.properties. Zdroj: Autor

```
user.label=Uživatel
user.login=Přihlášení
user.logout=Odhlásit
user.registration.message=Registrace proběhla úspěšně, můžete se
přihlásit.
```

5.8 Implementace testů

Poslední částí vývoje obou aplikací je tvorba automatizovaných testů. Oba dva frameworky nabízejí své vlastní, více či méně sofistikované testovací „frameworky“.

5.8.1 Framework Nette

Aplikace v Nette lze testovat nejběžněji pomocí sady knihoven určených pro testování PHPUnit, což je defacto standard v oblasti testování aplikací v jazyce PHP. Případně lze využít přibalenou jednoduchou sadu knihoven určenou protestování frameworku přímo od tvůrců. Rozhodl jsem se zvolit druhou cestu a testování aplikace demonstrovat právě v tomto prostředí.

[66] Seznam identifikátorů jednotlivých jazyků dle normy ISO 639-1 je dostupný zde: http://www.loc.gov/standards/iso639-2/php/English_list.php
[67] Seznam identifikátorů jednotlivých zemí dle normy ISO 3166-1 je dostupný zde: http://www.iso.org/iso/country_codes/iso_3166_code_lists/country_names_and_code_elements.htm

Pro začátek je třeba říci, že v případě použití těchto knihoven se nejedná o tzv. jednotkové testy, nýbrž o integrační. Dojde k inicializaci celého kontajneru a potažmo i celé aplikace tak, jako kdyby byla reálně spuštěná. Samozřejmě lze pomocí těchto knihoven testovat i jednotkově, ale netroufám si tvrdit, jaký by byl výsledek testování, konkrétně presenterů, bez použití mockovacích nástrojů.

Samotné testy jsou umístěné ve složce test a musí se při jejich psaní dodržovat konvence. Název souboru musí končit sufixem Test, název třídy musí dědit od třídy TestCase a každý jednotlivý test musí začínat prefixem test. Konstruktor takového testu má parametr ve formě kontajneru, který je vhodné si uchovat v privátní proměnné pro pozdější použití. V setUp fázi dojde k vytvoření presenteru a předání kontajneru do konstruktoru, tak vznikne plnohodnotná instance daného presenter připravená reagovat na testy.

U presenterů se testuje většinou typ odpovědi, přítomnost nějaké komponenty, či lze položit dotaz přímo do databáze, zda došlo k dané akci i tam. Testovat jiné části aplikace integračně a za předpokladu omezených nástrojů nemá příliš význam. Na testování prezentační vrstvy jsou lepší nástroje (Selenium) a testovat promítání změn z doménové vrsty do databáze není vždy triviální záležitostí. Uživatelé se velmi často spokojí s otestováním presenteru. Níže uvádím část kódu z testování presenteru user. Dojde k vytvoření requestu, což je v podstatě simulace odeslaného požadavku z prohlížeče a jeho předání presenteru a ověření výstupu. V tomto duchu probíhají ve své podstatě všechny ostatní testy.

Zdrojový kód 21 – Nette – testování neplatného prihlášení. Zdroj: Autor

```
function testLoginSubmitFail(){
    $request = new \Nette\Application\Request('User','POST',
        array(
            'action' => 'in',
            'do' => 'signInForm-submit'
        ),
        array(
            'email' => '',
            'password' => ''
        )
    );

    $response = $this->presenter->run($request);

    Assert::true(
        $response instanceof \Nette\Application\Responses\TextResponse
    );
}
```

5.8.2 Framework Grails

Testování je jednou z částí, kde je znát snaha autorů frameworku o velkou jednoduchost při dodržení velké komplexity. Troufám si říci, že testování je v Grails na velmi vysoké úrovni. Při testování jsou na výběr dvě možnosti, zda testovat jednotkově či integračně. Oproti Nette zde zvolím přístup pomocí jednotkového testování, abych demonstroval lépe to, co Nette chybí. Tím jsou mockovací nástroje. Možnosti testování jsou opravdu široké a lze otestovat nejen controller, ale i omezení modelu, mapování nebo filtry. Všech těchto možností jsem využil a ve zdrojových kódech přiložené aplikace se s nimi lze seznámit blíže. Dále pouze nastíním, jakým způsobem jsem provedl testování controlleru user.

Konvence jsou zde podobné jako v Nette, každá třída musí mít v názvu sufix Test nebo Tests. Soubor se, podobně jako v javě, jmenuje stejně, jako třída. Každý jednotlivý test pak začíná prefixem test. Třídě je třeba předat meta data ve formě anotace TestFor(Název třídy), která informuje prostředí o tom, která třída/controller bude testována. Automaticky je pak v jednotlivých testech k dispozici proměnná controller reprezentující instanci dané třídy, kterou testujeme. Doplňkově je možné zařadit anotaci Mock, která vytvoří tzv. mock objekt. Příkladem může být model User. Tento model lze pak používat stejným způsobem jako v controlleru, jako kdyby bylo spojení s databází skutečně aktivní. Logika testů je shodná s Nette. Testuje se reakce presenteru na určité vstupní podněty. Dochází k ověřování, zda jejich odpověď jako taková nebo i vrácené parametry jsou v souladu s požadavky, či zda se změny promítly do databáze.

Zdrojový kód 22 – Grails – testování neplatně a platně vyplněného registračního formuláře controlleru user. Zdroj: Autor

```
@TestFor(UserController)
@Mock(User)
class UserControllerTests {

    /**
     * Testování reakce při pokusu uložit špatně vyplněný registrační
     * formulář
     */
    void testRegistrationFormInvalid() {
        params.username = "pokus"
        params.passwordRegistration = "pokus"

        controller.save()

        assert model.userInstance != null
        assert view == '/user/registration'
    }
```

```
    /**
     * Testování funkčnosti registračního formuláře
     */
    void testRegistrationFormValid(){

        params.username = "pokus@pokus.cz"
        params.passwordRegistration = "pokus"
        params.passwordConfirm = "pokus"

        controller.save()

        assert response.redirectedUrl == '/user/login'
        assert flash.message != null
        assert User.count() == 1
        assert User.get(1).password ==
               params.passwordRegistration.encodeAsMD5()
    }
}
```

5.8.3 Srovnání

Testování je dnes nedílnou součástí vývoje, takže je znát i snaha tvůrců frameworků o jeho podporu. Diskutabilní však bývá míra této snahy. Grails jednoznačně na tomto poli dominuje, jeho podpora pro testování je na velmi vysoké úrovní. Poměrně snadno lze otestovat téměř jakoukoliv část aplikace. Oproti tomu Nette disponuje základní podporou, která i přes to, že stačí, by mohla být na vyšší úrovni. Minimálně zabudovanou podporu mock objektů by hodně vývojářů jistě uvítalo.

5.9 Zhodnocení vývoje u obou frameworků

Komentáře k jednotlivým částem jsem psal průběžně. Na závěr bych si dovolil zmínit ryze subjektivní názor a velmi stručně shrnout své poznatky z vývoje. Grails je podstatně komplikovanější i komplexnější framework, který si však poměrně dobře zachovává svůj jednoduchý přístup k vývoji. I přes tento fakt jsem při vývoji narazil na celou řadu problémů či faktů, které nebyly téměř vůbec dokumentovány (i přes to, že jinak existuje velice dobrá dokumentace) a bylo velice obtížné dopátrat se výsledku. Nette nemá zdaleka tak dobře zpracovanou dokumentaci jako Grails, ale poměrně dobře tento nedostatek supluje aktivní komunita s bohatými zdroji na oficiálním fóru.

Ani s jedním frameworkem jsem nenarazil na žádný neřešitelný problém, vývoj tedy musím hodnotit kladně. Pravdou však je, že u Grails jsem, obzvlášť při práci s GORM, občas narazil na velmi špatnou informační hodnotu u varovných hlášek. Netroufám si určit, který

framework je lepší či horší, protože potenciálu Grails, potažmo Groovy, jsem jako začátečník na této platformě pravděpodobně příliš nevyužil.

Za velice zásadní považuji ten fakt, že oba frameworky jsou velice odlišné, co se týče své mohutnosti či komplexnosti. Nette jde cestou čistoty kódu, návrhových vzorů a jednoduchosti s možností upravit si či změnit téměř cokoliv a velmi často je vývojář nucen psát i základní věci vlastními silami. Grails jde naopak cestou rubustnosti a komplexnosti. V základu Framework obsahuje prakticky všechny potřebné věci, které může vývojář potřebovat. Tímto bude pravděpodobně determinována i oblast použití. Grails se spíše hodí do podnikové sféry či na projekty většího rozsahu, a to i díky podpoře, kterou Nette de facto nemá. Použití Grails bude také velmi ovlivněno svými možnostmi kombinace s Javou a jejími knihovnami. Opomenout nelze ani rozdíl mezi platformou Java EE, potažmo Groovy/Grails a PHP. Nette bude tedy mířit pravděpodobně na aplikace menšího rozsahu či méně komplikované/komplexní než Grails.

V práci jsou z důvodu své rozsáhlosti zobrazovány pouze některé části zdrojových kódů aplikací. Kompletní znění obou aplikací ve formě zdrojových kódů, příkazů k vygenerování databáze a příručky pro zprovoznění je tedy k dispozici v Příloha 3 – Zdrojové kódy vytvořených aplikací.

6 Závěr

Prvním cílem této práce bylo stručně popsat framework Nette včetně jazyka PHP a framework Grails včetně jazyka Groovy. Po pečlivém nastudování základní terminologie a jejím stručném představení v úvodní kapitole jsem popsal ze široka obě platformy. Přiblížil jsem význam a základní vlastnosti dynamických jazyků, poté jsem stručně seznámil čtenáře s popularitou dynamických jazyků i aktuálními pozicemi obou jazyků dle všeobecně známého indexu popularity. Poté následovalo stručné představení charakteristických rysů obou jazyků i frameworků, včetně jejich historického vývoje.

Dalším cílem této práce bylo definovat takovou sadu kritérií, podle které bude možné dostatečně podrobně porovnávat frameworky pro rychlý vývoj aplikací. Na základě provedené analýzy se mi podařilo definovat sadu dvacetidvou kritérií pro následné srovnání. Tato kritéria nahlíží na framework komplexně a srovnání tedy proběhlo nejen na základě technicky orientovaných vlastností frameworků jako jsou například implementovaná architektura, práce s adresami, doménová vrstva, ale i netechnických jako jsou dostupné pracovní nabídky, kvalita dokumentace, dostupnost či podpora. Na frameworky bylo tedy nahlíženo velmi komplexně.

Definovaná kritéria jsem porovnal i mezi sebou z hlediska důležitosti metodou Fullerova trojúhelníku. Na základě těchto kritérií a stanovených vah jsem provedl vyhodnocení, včetně patřičných komentářů. Dle výsledků vyšel lépe Grails, což bylo způsobeno především díky lépe řešenému Dependency Injection kontajneru, lepší podpoře testování, kvalitnější dokumentaci, větších možnostech rozšíření a samozřejmě v existenci generátoru kódu. Závěrem bych však pouze zmínil, že oba frameworky měly velmi dobré hodnocení a domnívám se, že opodstatněný výběr jednoho, či druhého pravděpodobně nebude špatnou volbou ani v jednom případě.

Poslední částí práce a zároveň i cílem byl vývoj aplikace v obou prostředích a jeho srovnání. Před samotným vývojem jsem relativně podrobně navrhl aplikaci jak v obecném pohledu, tak konkrétním, dle daného implementačního prostředí. Na základě tohoto návrhu jsem vyvinul dvě funkčně totožné aplikace se zaměřením jako podpůrný prostředek metodiky GTD. Vývoj jsem rozčlenil do několika etap a opět jsem ho ilustrativně průběžně komentoval a dokumentoval fragmenty kódu. Jelikož je součástí kriteriálního hodnocení frameworků i kritérium hodnocení z praktického vývoje, zpětně jsem doplnil hodnoty do tabulky hodnocení jednotlivých implementačních kroků.

Cíle práce se mi tedy podařilo splnit a domnívám se, že práce může být dobrým východiskem pro vývojáře, kteří zvažují přechod z jedné vývojové platformy do druhé či se pouze rozhodují, jakou z platforem pro novou aplikaci zvolit. Taktéž spatřuji nezanedbatelný význam v definování ucelené sady kritérií pro ohodnocení RAD frameworků, která může být použita v dalších porovnáních.

Práce by mohla být dále rozšířena o hloubkovou analýzu výkonnosti, důkladný rozbor persistentní vrstvy, či o definování dalších kritérií.

7 Terminologický slovník

Termín	Zkratka	Popis
Object-relational mapping	ORM	Mapování objektů a vztahů na databázové tabulky a vztahy mezi nimi.
Keep it short and simple/stupid	KISS	Princip, který nabádá k psaní krátkého a jednoduchého kódu, který bude i snadno pochopitelný a udržovatelný.
Don't repeat yourself	DRY	Princip, který nabádá k neopakování stejného zdrojového kódu na více místech. Vychází z myšlenky, že každá dílčí znalost musí mít v systému jedinou, jednoznačnou, směrodatnou reprezentaci. Jsou však důvody, které dovolují toto pravidlo porušovat. [Jonáš, 2012]
Dependency Injection	DI	Podstatou DI je odebrat třídám zodpovědnost za získávání objektů, které potřebují ke své činnosti a místo toho jim instance těchto objektů předávat při vytváření. Existují různé druhy např. dle metody injekce setter, getter nebo constructor. [Grudl, 2012]
Common Gateway Interface	CGI	Rozhraní pro komunikaci webserveru s aplikací (CGI skriptem, napsaném například v shellu, Perlu, Pythonu…) [Kosek, 2012]
Duck typing		Jedná se o vlastnost jazyka, která říká, že pokud něco vypadá jako kachna a chodí jako kachna, pak to nejspíš bude kachna. V praxi to znamená, že se objekt nespoléhá na deklarovaný typ objektu, ale spoléhá na to, že odkazovaný objekt umí dodat implementaci dané metody. [Subramaniam, 2008]
Asynchronous Javascript and	AJAX/	Technologie původně vyvinutá firmou

Termín	Zkratka	Popis
XML / Asynchronous Javascript and JSON	AJAJ	Microsoft umožňující vysílat, resp. přijímat asynchronní požadavky bez nutnosti znovunačtení stránky. X nebo J na konci slova značí, že komunikace mezi skriptem a serverem je ve formátu XML nebo JSON.
Cache		V aplikačním světě se jedná o vyrovnávací paměť s různou implementací (filesystém, database). Cílem je zrychlit přístup k jinak pomaleji získávaným datům. [autor]
Java Virtual Machine	JVM	Programy napsané v Javě se kompilují do bytekódu, který je nezávislý na platformě. Konkrétní platforma má pak svoji implementaci JVM, který umí interpretovat aplikace z bytekódu. [Reese, 2012]
Grails Object-relational mapping	GORM	Objektově relační mapper postavený nad Hibernate uzpůsobený pro jazyk Groovy a framework Grails. [Judd, a další, 2008]
Java standard tag library	JSTL	Knihovna uměle vytvořených html značek, které zjednodušují a zpřehledňují vývoj díky možnosti vynechat skriplety a zapisovat jen tyto značky, které v sobě implementují nějakou z často se opakujících operací. [Rocher, a další, 2009]
Convention over Configuration	CoC	Přístup, kdy je preferováno dodržování konvenčních nařízení před explicitním definováním standardu.
Scaffolding		Funkcionalita některých frameworků, která dovoluje rychle vygenerovat CRUD interface k existujícímu datovému modelu. [Rocher, a další, 2009]
Create, Read, Update, Delete	CRUD	Zkratka pro operace nad modelem (doménovým) vyjadřující čtyři základní

Termín	Zkratka	Popis
		oprace, a to konkrétně vytváření, čtení, modifikace a mazání.
Unique Resource Locator	URL	Je to způsob, jak jednoznačně zapsat umístění souboru na Internetu nebo na intranetu. URL je synonymem pro internetové adresy. [Janovský, 2012]
SQL Inejction		Druh útoku na aplikace spočívající ve spuštění útočníkova kódu v databázi aplikace díky špatně ošetřeným vstupům.
Cross-Site Request Forgery	CSRF	Druh útoku na webovou aplikaci, který spočívá v podvržení http požadavku s cílem získat informace nebo provést nějakou akci. Když jsme přihlášeni např. v našem redakčním systému (útočník ho zná a ví, jakým způsobem se volají funkce pro smazání uživatele) a máme oprávnění na smazání určité osoby, stačí útočníkovi, aby nás donutil (například v e-mailu) zobrazit obrázek, který bude mít jako zdroj uvedenou adresu na smazání uživatele. [Frank, 2011]
YAML	YAML	Populární formát pro serializaci strukturovaných dat. Oproti XML je úsporný, ale zároveň přehledný a strojově velmi snadno čitelný. Nejedná se o značkovací jazyk. [Evans, 2011]
JavaScript Object Notation	JSON	Populární formát pro serializaci strukturovaných dat používaný často pro jejich přenos. Zjednodušeně se jedná o speciální zápis pole typu klíč hodnota se zanořováním. [Crockford]
Wireframe		V oblasti vývoje webových prezentací či aplikací se používá pro zjednodušený model

Termín	Zkratka	Popis
		či architektonický návrh definující funkce a obsah stránek webu. [Symbio, 2013]

8 Použité zdroje

Allen, David. 2002. *Getting Things Done - The Art of Stress-Free Productivity.* New York : Penguin Books, 2002. 978-0142000281.

Bernard, Bořek. 2009. Seriál MVC a další prezentační vzory. *Zdroják.cz.* [Online] Internet Info s.r.o. , 2009. [Cited: prosinec 3, 2012.] http://www.zdrojak.cz/serialy/mvc-a-dalsi-prezentacni-vzory/.

Buchalcevová, Alena, Pavlíček, Luboš and Pavlíčková, Jarmila. 2007. *Základy softwarovéh inženýrství - materiály ke cvičení.* Praha : VŠE - nakladatelství Oeconomica, 2007. 978-80-245-1270-9.

Cake Software Foundation Inc. 2013. The rapid development php framework. *CakePHP.* [Online] Cake Software Foundation Inc., 2013. [Cited: březen 12, 2013.] http://www.cakephp.org/.

Clifton, Marc. 2003. What Is A Framework? *CodeProject.* [Online] CodeProject, listopad 3, 2003. [Cited: listopad 30, 2012.] http://www.codeproject.com/Articles/5381/What-Is-A-Framework.

Coyier, Chris. 2010. Poll Results: Server side language of choice? *CSS Tricks.* [Online] květen 26, 2010. [Cited: leden 25, 2013.] http://css-tricks.com/poll-results-server-side-language/.

Crockford, Daniel. *Introducing JSON.* [Online] [Cited: březen 9, 2013.] http://www.json.org/.

Evans, Clark. 2011. *The offical YAML website.* [Online] listopad 2011, 2011. [Cited: březen 7, 2013.] http://www.yaml.org/.

Florian, Jan. 2011. *Porovnání frameworků pro rychlý vývoj aplikací Dajngo a Rails.* Praha : VŠE, 2011.

Fowler, Martin. 2005. Inverision of Control. *Martin Fowler.* [Online] červen 25, 2005. [Cited: březen 13, 2013.] http://martinfowler.com/bliki/InversionOfControl.html.

—. 2004. Inversion of Control Containers and the Dependency Injection pattern. *Martin Fowler.* [Online] leden 24, 2004. [Cited: březen 8, 2013.] http://martinfowler.com/articles/injection.html.

Fowler, Martin, et al. 2002. *Patterns of Enterprise Application Architecture.* Boston : Addison Wesley, 2002. 0-321-12742-0.

Frank, Jiří. 2011. *Audit webových stránek internetových obchodů.* Praha : VŠE, 2011.

Grudl, David. 2012. CO JE DEPENDENCY INJECTION? *PHPFashion.* [Online] březen 12, 2012. [Cited: březen 1, 2013.] http://phpfashion.com/co-je-dependency-injection.

—. 2012. Dokumentace. *Nette.* [Online] Nette foundation, 2012. [Cited: listopad 23, 2012.] http://doc.nette.org/cs/.

Hanyš, Pavel. 2010. *Srovnání javascriptových frameworků.* Praha : VŠE, 2010.

Hradecký, Radim. 2009. *Srovnání softwarových frameworků.* Praha : ČVUT, 2009.

Janovský, Dušan. 2012. *Jak psát web.* [Online] prosinec 4, 2012. [Cited: prosinec 9, 2012.] http://www.jakpsatweb.cz/.

Jonáš, Martin. 2012. Návrhové principy. *Zdroják.cz.* [Online] Devel.cz Lab s.r.o., květen 9, 2012. [Cited: prosinec 1, 2012.] http://www.zdrojak.cz/clanky/navrhove-principy-solid/.

Judd, Christopher M., et al. 2008. *Beginning Groovy and Grails: From Novice to Professional.* New York : Apress, 2008. 978-1430210450.

Kosek, Jiří. 2012. Přednášky. *4IZ228.* [Online] 2012. [Cited: prosinec 2, 2012.] http://webovky.vse.cz/vyuka/prednasky.

Křivánek, Pavel. 2004. http://www.root.cz/clanky/staticka-dynamicka-typova-kontrola/. *Root.cz.* [Online] červen 24, 2004. [Cited: listopad 20, 2012.] http://www.root.cz/clanky/staticka-dynamicka-typova-kontrola/.

L'institut d'électronique et d'informatique Gaspard-Monge. 2009. Groovy and Grails - Architecture. *L'institut d'électronique et d'informatique Gaspard-Monge.* [Online] 2009. [Cited: listopad 28, 2012.] http://www-igm.univ-mlv.fr/~dr/XPOSE2009/Groovy_and_Grails/grails_architecture.php.

Macintire, Peter, Danchilla, Brian and Gogala, Mladen. 2011. *Pro PHP.* New York : APress, 2011. 978-1430235606.

Marc, Palmer. 2012. Native will beat web apps on desktop too. *Marc Palmer´s blog.* [Online] červenec 27, 2012. [Cited: listopad 30, 2012.] http://www.anyware.co.uk/2005/2011/07/27/native-will-beat-web-apps-on-desktop-too/.

Metzner, Ryan. 2012. Why Web Apps Will Crush Native Apps. *Mashable Tech.* [Online] Mashable, Inc., září 12, 2012. [Cited: listopad 30, 2012.] http://mashable.com/2012/09/12/web-vs-native-apps/.

Microsoft. 2011. Úvod k přepisování adres URL, k mapování adresy URL a URL směrování technologie ASP.NET v prostředí rozhraní.NET Framework. *Microsoft.* [Online] Microsoft, květen 24, 2011. [Cited: březen 13, 2013.] http://support.microsoft.com/kb/976111.

Odehnal, František. 2011. Komplexní srovnání vývoje webových aplikací ve frameworcích Zend (PHP) a Grails (Java-Groovy). *ISIS.* [Online] 2011. [Cited: listopad 5, 2012.] https://isis.vse.cz/auth/lide/clovek.pl?zalozka=7;id=50681;studium=109749;download_prace=1;lang=cz.

Olišar, Petr. 2012. *Porovnání Nette a Zend frameworku.* Praha : VŠE, 2012.

Oliver, Plohmann. 2012. Performance Test: Groovy 2.0 vs. Java. *DZone.* [Online] DZone, Inc., srpen 28, 2012. [Cited: listopad 25, 2012.] http://java.dzone.com/articles/groovy-20-performance-compared.

Parr, Terence. 2004. *Enforcing strict model-view separation in template engines.* [Dokument] New York : University of San Francisco, 2004. 1-58113-844-X.

Pecinovský, Rudolf. 2012. Architektonické vzory podnikových aplikací. *Pecinovský - Objektově orientované programování a jeho výuka.* [Online] listopad 26, 2012. [Cited: únor 19, 2013.] http://vyuka.pecinovsky.cz/vse/422/Architektonicke_vzory.pdf.

PHPZag. 2012. Top 5 PHP Frameworks in 2012. *PHPZag.* [Online] květen 30, 2012. http://www.phpzag.com/top-5-php-frameworks-2012/.

Pierce, C. Benjamin. 2002. *Types and Programming Languages.* Cambridge, Massachusetts : The MIT Press, 2002. 0-262-16209-1.

Pichlík, Roman. 2007. Chudokrevné a plnokrevné IoC, EJB vs. Spring. *Dagblog.* [Online] leden 10, 2007. [Cited: únor 25, 2013.] http://www.dagblog.cz/2007/01/chudokrevn-plnokrevn-ioc-ejb-vs.html.

Potencier, Fabien. 2009. Templating Engines in PHP. *fabien.potencier.org.* [Online] říjen 7, 2009. [Cited: březen 19, 2013.] http://fabien.potencier.org/article/34/templating-engines-in-php.

Rails Core Team. 2013. *Ruby on Rails.* [Online] únor 13, 2013. [Cited: únor 25, 2013.] http://rubyonrails.org/.

RedHat. 2012. Hibernate. *jBoss Community.* [Online] RedHat, 2012. [Cited: listopad 2, 2012.] http://www.hibernate.org/.

Reese, Richard M. 2012. *Oracle Certified Associate, Java SE 7 Programmer Study Guide.* Birmingham : s.n., 2012. 978-1-84968-732-4.

Rocher, Graeme and Brown, Jeff. 2009. *The Definitive Guide to Grails.* New York : APress, 2009. 978-1-4302-0871-6.

Spring Source. 2012. Documentation. *Groovy.* [Online] 2012. [Cited: listopad 18, 2012.] http://groovy.codehaus.org/Documentation.

—. 2012. The search is over. *Grails.org.* [Online] Spring Source, 2012. [Cited: listopad 25, 2012.] http://www.grails.org/.

—. 2012. Version history. *Groovy.* [Online] Spring Source , 2012. [Cited: listopad 26, 2012.] http://groovy.codehaus.org/History.

SpringSource. 2012. Spring documentation. *SpringSource.* [Online] SpringSource, 2012. [Cited: prosinec 2, 2012.] http://www.springsource.org/documentation.

Subramaniam, Venkat. 2008. *Programming Groovy.* Dallas : The Pragmatic Bookshelf, 2008. 1-934356-09-3.

Symbio. 2013. Slovník - Wireframe webu. *Symbio.* [Online] SYMBIO Digital, s. r. o, 2013. [Cited: duben 1, 2013.] http://www.symbio.cz/slovnik/wireframe-webu.html.

The PHP Group. 2012. *PHP: Hypertext preprocessor.* [Online] The PHP Group, listopad 23, 2012. [Cited: listopad 23, 2012.] http://www.php.net.

Tiobe Software. 2013. Tiobe Index. *Tiobe Software.* [Online] Tiobe Software, duben 2013. [Cited: duben 1, 2012.] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Towards the Rapid Application Development. **Yu, Dongjin. 2011.** 9, Hangzhou : Hangzhou Dianzi University, 2011, Vol. VI.

Válek, Ondřej. 2008. *Webové aplikační frameworky pro rychlý vývoj aplikací.* Praha : VŠE, 2008.

VersionOne. 2010. The State of Agile Development Survey Results. *VersionOne.com.* [Online] 2010. [Cited: listopad 2, 2012.] http://www.versionone.com/pdf/2010_State_of_Agile_Development_Survey_Results.pdf.

Vrána, Jakub. 2011. *1001 tipů a triků pro PHP.* Praha : CPress, 2011. 978-80-251-2940-1.

W3Techs. 2012. Usage statistics and market share of PHP version 5 for websites. *W3Techs - Web technology serveys.* [Online] Q-Success, listopad 2012. [Cited: listopad 24, 2012.] http://w3techs.com/technologies/details/pl-php/5/all.

Weyner, Peter. 2011. From PHP to Perl: What's hot, what's not in scripting languages. *InfoWorld.com.* [Online] říjen 17, 2011. [Cited: listopad 21, 2012.] http://search.proquest.com.ezproxy.vse.cz/docview/898612360?accountid=17203.

Wikipedia. 2012. Groovy (programming language). *Wikipedia.* [Online] Wikimedia Foundation, Inc., listopad 12, 2012. [Cited: listopad 25, 2012.] http://en.wikipedia.org/wiki/Groovy_(programming_language).

Zend Technologies. 2013. *Zend Framework.* [Online] 2013. [Cited: březen 12, 2013.] http://framework.zend.com/.

Zilvar, Tomáš. 2010. *Frameworky pro rychlý vývoj webových aplikací na platformě Java.* Praha : VŠE, 2010.

Zorgdrager, Kelby. 2010. Choosing the Right Java Web Development Framework. *OpenLogic.* [Online] OpenLogic, Inc., červenec 29, 2010. http://www.openlogic.com/wazi/bid/188143/Choosing-the-Right-Java-Web-Development-Framework.

9 Seznamy

9.1 Tabulky

Tabulka 1 – Vyhodnocení kritéria dostupnost. Zdroj: Autor 21
Tabulka 2 – Vyhodnocení kritéria podpora. Zdroj: Autor 21
Tabulka 3 – Vyhodnocení kritéria architektura frameworku. Zdroj: Autor 22
Tabulka 4 – Vyhodnocení kritéria práce s adresami. Zdroj: Autor 24
Tabulka 5 – Vyhodnocení kritéria Convention over Configuration. Zdroj: Autor 24
Tabulka 6 – Vyhodnocení podkritérií kritéria Dependency Injection. Zdroj: Autor 27
Tabulka 7 – Vyhodnocení kritéria Dependency Injection. Zdroj: Autor 27
Tabulka 8 – Vyhodnocení podkritérií podpory testování. Zdroj: Autor 28
Tabulka 9 – Vyhodnocení kritéria podpora testování. Zdroj: Autor 28
Tabulka 10 – Vyhodnocení kritéria integrované funkce v základu. Zdroj: Autor 29
Tabulka 11 – Vyhodnocení kritéria integrované funkce v základu. Zdroj: Autor 30
Tabulka 12 – Přehled dostupných běhových prostředí. Zdroj: Autor 31
Tabulka 13 – Přehled dostupných běhových prostředí pro Nette. Zdroj: Autor 31
Tabulka 14 – Vyhodnocení kritéria dostupné běhové prostředí. Zdroj: Autor 32
Tabulka 15 – Vyhodnocení kritéria AJAX/AJAJ. Zdroj: Autor 33
Tabulka 16 – Podpora databázových systému oběma frameworky. Zdroj: Autor 36
Tabulka 17 – Vyhodnocení podkritérií doménové vrstvy. Zdroj: Autor 36
Tabulka 18 – Vyhodnocení kritéria persistentní a doménová vrstva. Zdroj: Autor 37
Tabulka 19 – Vyhodnocení podkritérií pro zhodnocení kritéria šablonování. Zdroj: Autor 38
Tabulka 20 – Vyhodnocení kritéria šablonování. Zdroj: Autor 39
Tabulka 21 – Zhodnocení jednotlivých podkritérií pro možnost rozšíření. Zdroj: Autor 40
Tabulka 22 – Vyhodnocení kritéria možnosti rozšíření. Zdroj: Autor 41
Tabulka 23 – Shrnutí možností lokalizací. Zdroj: Autor 42
Tabulka 24 – Vyhodnocení kritéria podpora lokalizací. Zdroj: Autor 42
Tabulka 25 – Přehled základních podporovaných funkcí. Zdroj: Autor 43
Tabulka 26 – Vyhodnocení kritéria generátory. Zdroj: Autor 44
Tabulka 27 – Hodnocení lokálního a globálního rozšíření Zdroj: Autor 46
Tabulka 28 – Vyhodnocení kritéria rozšířenost a význam. Zdroj: Autor 46
Tabulka 29 – Srovnání podpory pro práci s frameworky v některých IDE. Zdroj: Autor 47
Tabulka 30 – Vyhodnocení kritéria integrované vývojové prostředí. Zdroj: Autor 47

Tabulka 31 – Subjektivní hodnocení z vývoje aplikace Zdroj: Autor 48
Tabulka 32 – Vyhodnocení kritéria hodnocení z praktického vývoje. Zdroj: Autor 48
Tabulka 33 – Vyhodnocení kritéria aktivní vývoj. Zdroj: Autor 50
Tabulka 34 – Výchozí hodnoty parametrů pro výpočet indikátorů aktivity. 51
Tabulka 35 – Hodnoty indikátorů aktivity komunity. Zdroj: Autor 52
Tabulka 36 – Vyhodnocení kritéria aktivní komunita. Zdroj: Autor 52
Tabulka 37 – Vyhodnocení kritéria dokumentace. Zdroj: Autor 53
Tabulka 38 – Počty nabídek zaměstnání vybraných pracovních portálů k 24.3.2013 54
Tabulka 39 – Vyhodnocení kritéria pracovní nabídky. Zdroj: Autor 54
Tabulka 40 – Závěrečná sumarizační tabulka všech kritérií. Zdroj: Autor 55
Tabulka 41 – Případ užití – Registrace uživatele. Zdroj: Autor 61
Tabulka 42 – Případ užití – Přihlášení uživatele. Zdroj: Autor 62
Tabulka 43 – CRC karta - Uživatel. Zdroj: Autor 63
Tabulka 44 – CRC karta - Úkol. Zdroj: Autor 63
Tabulka 45 – CRC karta – Stav úkolu. Zdroj: Autor 64
Tabulka 46 - CRC karta - Značka. Zdroj: Autor 64
Tabulka 47 - CRC karta - Komentář. Zdroj: Autor 64
Tabulka 48 – Případ užití – Odhlášení uživatele. Zdroj: Autor 116
Tabulka 49 - Případ užití – Vytvoření nového úkolu. Zdroj: Autor 116
Tabulka 50 - Případ užití – Upravit úkol. Zdroj: Autor 117
Tabulka 51 – Případ užití – Zobrazit detail úkolu. Zdroj: Autor 117
Tabulka 52 - Případ užití – Smazání úkolů. Zdroj: Autor 118
Tabulka 53 - Případ užití – Prohledávání úkolů. Zdroj: Autor 118
Tabulka 54 - Případ užití – Nový komentář. Zdroj: Autor 119
Tabulka 55 - Případ užití – Smazat komentář. Zdroj: Autor 119
Tabulka 56 – Případ užití – Přidání nového tagu. Zdroj: Autor 120
Tabulka 57 – Případ užití – Úprava stávajícího tagu. Zdroj: Autor 120
Tabulka 58 – Případ užití – Smazání tagu. Zdroj: Autor 121

9.2 Obrázky

Obrázek 1 – Jedna z možných implementací MVC. Zdroj: Autor dle [Bernard, 2009] 7
Obrázek 2 – Diagram MVP. Zdroj: Autor dle [Bernard, 2009] 8
Obrázek 3 – Implementace MVP v Passive View. Zdroj: Autor dle [Bernard, 2009] 8

Obrázek 4- Tiobe Index. Zdroj: [Tiobe Software, 2013]..... 11
Obrázek 5 – Relativní zájem o Grails a Nette letech 2006 až 2013. Zdroj: Google 44
Obrázek 6 – Mapa zobrazující zdrojové státy hledání Grails. Zdroj: Google..... 45
Obrázek 7 - Mapa zobrazující zdrojové státy hledání Nette. Zdroj: Google 45
Obrázek 8 - Relativní zájem o Grails a Nette v letech 2006 až 2013 v ČR. Zdroj: Google 45
Obrázek 9 – Frekvence commitů zdrojových souborů Grails. Zdroj: Github..... 49
Obrázek 10 – Frekvence commitů zdrojových souborů Nette. Zdroj: Github..... 49
Obrázek 11 – Diagram případů užití. Zdroj: Autor 61
Obrázek 12 – Statický diagram tříd. Zdroj: Autor 65
Obrázek 13 – Wireframe - Přihlášení. Zdroj: Autor..... 67
Obrázek 14 – Wireframe - Registrace. Zdroj: Autor..... 67
Obrázek 15 – Wireframe – Hlavní strana. Zdroj: Autor..... 68
Obrázek 16 – Architektura aplikace ve frameworku Grails. Zdroj: Autor..... 70
Obrázek 17 – Architektura aplikace ve frameworku Nette. Zdroj: Autor..... 71
Obrázek 18 – Nette - Entitně-relační diagram databáze aplikace. Zdroj: Autor 81
Obrázek 19 – Grails - Entitně-relační diagram databáze aplikace. Zdroj: Autor 84

9.3 Zdrojové kódy

Zdrojový kód 1 – Nette - Konfigurační soubor config,neon. Zdroj: Autor..... 77
Zdrojový kód 2 – Nette - Zaváděcí soubor bootstrap.php. Zdroj: Autor 78
Zdrojový kód 3 – Nette – Nastavení routování v souboru RouterFactory.php. Zdroj: Autor .. 79
Zdrojový kód 4 – Zaváděcí soubor bootstrap.groovy frameworku Grails. Zdroj: Autor..... 79
Zdrojový kód 5 – Grails - Nastavení routování UrlMappings.groovy. Zdroj: Autor 79
Zdrojový kód 6 – Grails – Nastavení datových zdrojů DataSource.groovy. Zdroj: Autor..... 80
Zdrojový kód 7 – Nette – Model User. Zdroj: Autor 82
Zdrojový kód 8 – Grails – Model User. Zdroj: Autor 83
Zdrojový kód 9 – Nette – Presenter User – injektáž závislostí. Zdroj: Autor 86
Zdrojový kód 10 – Nette – Presenter User – registrační formulář. Zdroj: Autor..... 87
Zdrojový kód 11 – Nette – UserPresenter – přihlašování uživatelů. Zdroj: Autor..... 88
Zdrojový kód 12 – Grails – UserController –metody pro přístup k akcím. Zdroj: Autor..... 89
Zdrojový kód 13 – Grails – UserController – registrace uživatelů. Zdroj: Autor..... 89
Zdrojový kód 14 – Grails – UserController – Přihlašování uživatelů. Zdroj: Autor..... 89

Zdrojový kód 15 – Nette – Šablona layout.latte. Zdroj: Autor 91
Zdrojový kód 16 – Nette – šablona user/reg.ctp. Zdroj: Autor 92
Zdrojový kód 17- Grails – layouts/main.gsp. Zdroj: Autor..... 92
Zdrojový kód 18 - Grails – user/registration.gsp. Zdroj: Autor 93
Zdrojový kód 19 - Grails – user/_form.gsp. Zdroj: Autor..... 95
Zdrojový kód 20 – Grails – lokalizační soubor message_cs_cz.properties. Zdroj: Autor 96
Zdrojový kód 21 – Nette – testování neplatného prihlášení. Zdroj: Autor 97
Zdrojový kód 22 – Grails – testování registračního formuláře. Zdroj: Autor 98

10 Příloha 1 – Případy užití

Tabulka 48 – Případ užití – Odhlášení uživatele. Zdroj: Autor

Název případu užití	Odhlášení uživatele
Identifikace	UC3
Cíl	Odhlášený uživatel
Primární aktér	Uživatel
Vstupní podmínky	Uživatel je přihlášen
Výstupní podmínky	Uživatel není přihlášen
Základní scénář	

Krok	**Role**	**Akce**
1	Uživatel	Klikne na tlačítko odhlásit.
2	Aplikace	Systém odhlásí uživatele a provede přesměrování na hlavní stranu s hláškou o úspěšném odhlášení.

Tabulka 49 - Případ užití – Vytvoření nového úkolu. Zdroj: Autor

Název případu užití	Vytvoření nového úkolu
Identifikace	UC5
Cíl	Nový úkol
Primární aktér	Uživatel
Vstupní podmínky	Uživatel je přihlášen
Výstupní podmínky	Vytvořený úkol
Základní scénář	

Krok	**Role**	**Akce**
1	Uživatel	Klikne na tlačítko nový úkol.
2	Aplikace	Zobrazí formulář s políčky pro vyplnění úkolu.
3	Uživatel	Vyplní hodnoty
4	Aplikace	Zkontroluje vyplněné hodnoty, pokud jsou chybně, zobrazí chybovou hlášku a zobrazí vyplněný formulář s označenými chybnými políčky. V opačném vytvoří nový úkol, provede přesměrování na přehled a zobrazí

			hlášku o úspěchu.

Tabulka 50 - Případ užití – Upravit úkol. Zdroj: Autor

Název případu užití	Upravit úkol
Identifikace	UC6
Cíl	Upravený úkol
Primární aktér	Uživatel
Vstupní podmínky	Uživatel je přihlášen, Úkol existuje, Úkol patří danému uživateli
Výstupní podmínky	Úkol byl upraven
Základní scénář	

Krok	**Role**	**Akce**
1	Uživatel	Vybere požadovaný úkol k úpravě a klikne na tlačítko upravit.
2	Aplikace	Zobrazí editační formulář úkolu.
3	Uživatel	Upraví hodnoty
4	Aplikace	Zkontroluje vyplněné hodnoty, pokud jsou chybně, zobrazí chybovou hlášku a zobrazí vyplněný formulář s označenými chybnými políčky. V opačném případě aktualizuje hodnoty, přesměruju na přehled úkolů a zobrazí hlášku, že se akce zdařila.

Tabulka 51 – Případ užití – Zobrazit detail úkolu. Zdroj: Autor

Název případu užití	Zobrazit úkol
Identifikace	UC6
Cíl	Zobrazený úkol
Primární aktér	Uživatel
Vstupní podmínky	Uživatel je přihlášen, Úkol existuje, Úkol patří danému uživateli
Výstupní podmínky	Úkol byl zobrazen
Základní scénář	

Krok	**Role**	**Akce**
1	Uživatel	Vybere požadovaný úkol k zobrazení a klikne na tlačítko symbolizující detail

	2	Aplikace	Zobrazí detail úkolu s přidruženými komentáři

Tabulka 52 - Případ užití – Smazání úkolů. Zdroj: Autor

Název případu užití	Smazat úkol		
Identifikace	UC7		
Cíl	Smazaný úkol		
Primární aktér	Uživatel		
Vstupní podmínky	Uživatel je přihlášen, Úkol existuje, Úkol patří danému uživateli		
Výstupní podmínky	Úkol byl smazán		
Základní scénář	**Krok**	**Role**	**Akce**
	1	Uživatel	Uživatel vybere požadovaný úkol a klikne na tlačítko smazat.
	2	Aplikace	Smaže úkol, provede přesměrování na přehled úkolů a zobrazí hlášku o úspěchu. V opačném případě pouze provede přesměrování na přehled úkolů.

Tabulka 53 - Případ užití – Prohledávání úkolů. Zdroj: Autor

Název případu užití	Prohledat úkoly		
Identifikace	UC9		
Cíl	Vyhledaný úkol		
Primární aktér	Uživatel		
Vstupní podmínky	Uživatel je přihlášen		
Výstupní podmínky	Požadované zobrazení úkolů		
Základní scénář	**Krok**	**Role**	**Akce**
	1	Aplikace	Zobrazuje přehled všech úkolů s možností akcí, stránkováním, řazením a filtrováním.
Varianta scénáře	**Krok**	**Role**	**Akce**
	1	Aplikace	Zobrazuje přehled všech úkolů s možností akcí a stránkováním, s možností vyhledávat

		a filtrovat
2	Uživatel	Zvolí kritéria a klikne na tlačítko aktualizovat
3	Aplikace	Zobrazí aktualizovaný seznam úkolů

Tabulka 54 - Případ užití – Nový komentář. Zdroj: Autor

Název případu užití	Nový komentář
Identifikace	UC11
Cíl	Přidaný nový komentář
Primární aktér	Uživatel
Vstupní podmínky	Uživatel je přihlášen, Úkol, ke kterému se vztahuje komentář, patří danému uživateli
Výstupní podmínky	Komentář je přidán
Základní scénář	

Krok	**Role**	**Akce**
1	Uživatel	Vyplní formulář pro přidání komentáře v detailu úkolů a stiskne tlačítko odeslat
4	Aplikace	V případě, že bylo políčko vyplněno v pořádku, uloží ho. Poté přesměruje uživatele zpět na detail daného úkolu a zobrazí hlášku o úspěšném uložení.

Tabulka 55 - Případ užití – Smazat komentář. Zdroj: Autor

Název případu užití	Smazat komentář
Identifikace	UC12
Cíl	Smazaný komentář
Primární aktér	Uživatel
Vstupní podmínky	Uživatel je přihlášen, Úkol, ke kterému se vztahuje komentář, patří danému uživateli
Výstupní podmínky	Komentář je smazán
Základní scénář	

Krok	**Role**	**Akce**
1	Uživatel	Klikne u komentáře na tlačítko

			symbolizující smazání komentáře.
	2	Aplikace	Smaže komentář, zobrazí hlášku o úspěšném smazání a provede přesměrování na detail úkolu.

Tabulka 56 – Případ užití – Přidání nového tagu. Zdroj: Autor

Název případu užití	Přidání nového tagu		
Identifikace	UC12		
Cíl	Přidání nového tagu		
Primární aktér	Uživatel		
Vstupní podmínky	Uživatel je přihlášen		
Výstupní podmínky	Tag je přidán		
Základní scénář	**Krok**	**Role**	**Akce**
	1	Uživatel	Klikne na tlačítko přidat nový tag ve sloupci tagy
	2	Aplikace	Přejde na stránku s formulářem
	3	Uživatel	Vyplní hodnoty
	4	Aplikace	Zkontroluje vyplněné hodnoty, pokud jsou chybně, zobrazí chybovou hlášku a zobrazí vyplněný formulář s označenými chybnými políčky. V opačném případě vytvoří nový úkol, provede přesměrování na přehled a zobrazí hlášku o úspěchu.

Tabulka 57 – Případ užití – Úprava stávajícího tagu. Zdroj: Autor

Název případu užití	Úprava stávajícícho tagu
Identifikace	UC12
Cíl	Upravený tag
Primární aktér	Uživatel
Vstupní podmínky	Uživatel je přihlášen, Tag existuje, Tag patří danému uživateli
Výstupní podmínky	Tag je upraven

Základní scénář	**Krok**	**Role**	**Akce**
	1	Uživatel	Klikne na symbol značící úpravu u požadovaného tagu
	2	Aplikace	Zobrazí editační formulář tagu
	3	Uživatel	Vyplní hodnoty
	4	Aplikace	Zkontroluje vyplněné hodnoty, pokud jsou chybně, zobrazí chybovou hlášku a zobrazí vyplněný formulář s označenými chybnými políčky. V opačném případě vytvoří nový úkol, provede přesměrování na přehled a zobrazí hlášku o úspěchu.

Tabulka 58 – Případ užití – Smazání tagu. Zdroj: Autor

Název případu užití	Smazání tagu		
Identifikace	UC12		
Cíl	Smazaný tag		
Primární aktér	Uživatel		
Vstupní podmínky	Uživatel je přihlášen, Tag existuje, Tag patří danému uživateli		
Výstupní podmínky	Tag je smazán		
Základní scénář	**Krok**	**Role**	**Akce**
	1	Uživatel	Klikne na symbol značící smazání u požadovaného tagu
	2	Aplikace	Smaže komentář, zobrazí hlášku o úspěšném smazání a provede přesměrování na stránku s přehledem úkolů.

11 Příloha 2 - Stanovení vah jednotlivých kritérií

Tato příloha je kvůli svým velkým rozměrům dostupná na přiloženém CD v elektronické podobě.

12 Příloha 3 – Zdrojové kódy vytvořených aplikací

Na přiloženém CD jsou uloženy kompletní zdrojové kódy obou aplikací. Pro každou z nich jsou tedy k dispozici zdrojové soubory ve složce aplikace, opis příkazů pro vytvoření databáze je ve složce databáze a stručný návod na zprovoznění aplikace ve složce příručka.