

# Návrh architektury internetového portálu fotbal.cz

Petr Hlaváč

---

Diplomová práce  
2016

 Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky

---

## ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Petr Hlaváč**  
Osobní číslo: **A14463**  
Studijní program: **N3902 Inženýrská informatika**  
Studijní obor: **Informační technologie**  
Forma studia: **kombinovaná**

Téma práce: **Návrh architektury internetového portálu fotbal.cz**  
Téma anglicky: **A Draft Architecture for the fotbal.cz Internet Portal**

### Zásady pro vypracování:

1. Identifikujte klíčové funkční a nefunkční požadavky na portál fotbal.cz.
2. Navrhněte architekturu webového portálu tak, aby splňovala stanovené požadavky, nalezněte vhodná technická řešení a odůvodněte jejich použití.
3. V teoretické části stručně popište technologie využité v rámci návrhu.
4. Popište klíčové fáze praktické implementace jednotlivých částí portálu.
5. Zhodnoťte přínosy i další aspekty návržné architektury.

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **MARTIN, Robert C.** Clean code: a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall, c2009, xxix, 431 p. ISBN 0132350882.
2. **HASIN HAYDER.** Object-oriented programming with PHP5 learn to leverage PHP5's features to write manageable applications with ease. Birmingham: Packt Pub, 2007. ISBN 1847192572.
3. **ANDREW CURIOSO, Ronald Bradford.** Expert PHP and MySQL. Indianapolis, IN: Wiley Pub, 2010. ISBN 047088164x.
4. **BY STEFAN JABLONSKI, Ilija Petrov.** Guide to Web Application and Platform Architectures. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN 3662076314.
5. **FOWLER, Martin.** Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003, xxiv, 533 s. Addison-Wesley signature series. ISBN 0-321-12742-0.

Vedoucí diplomové práce:

**Ing. Radek Vala, Ph.D.**

Ústav informatiky a umělé inteligence

Datum zadání diplomové práce:

**5. února 2016**

Termín odevzdání diplomové práce:

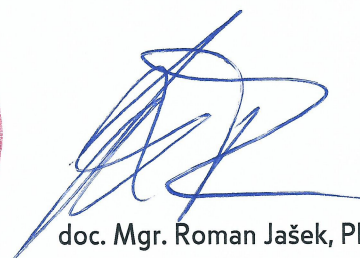
**20. května 2016**

Ve Zlíně dne 5. února 2016



doc. Mgr. Milan Adámek, Ph.D.

*děkan*



doc. Mgr. Roman Jašek, Ph.D.

*ředitel ústavu*

**Jméno, příjmení:** Petr Hlaváč

**Název bakalářské/diplomové práce:** Návrh architektury internetového portálu fotbal.cz

**Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji,**

- že jsem na diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 3.5.2016

.....  
podpis diplomanta



## **ABSTRAKT**

Práce se zabývá návrhem architektury internetového portálu fotbal.cz, přičemž důraz je kladen na rychlou a jednoduchou údržbu a aktualizaci aplikační i datové vrstvy. V teoretické části jsou popsány použité technologie. Praktická část je věnována rozboru zadání a návrhu vhodného řešení. Je předložen a diskutován návrh řešení dílčích problémů, jako je výběr vhodných technologií a architektury. Na základě těchto požadavků jsou popsány klíčové části procesu implementace. V závěru práce je provedeno zhodnocení navrženého řešení.

Klíčová slova: Internetový portál, Návrh architektury, PHP, Nette

## **ABSTRACT**

This thesis deals with the architecture of the Internet portal fotbal.cz. It deals with an emphasis on quick, easy maintenance, updating of application and data layers. The theoretical part describes the used technologies. The practical part is devoted to the analysis task and design appropriate solutions. It is presented and discussed a proposal addressing specific issues such as the selection of appropriate technologies and architectures. Based on these requirements are described key part of the implementation process. The conclusion is an assessment of the proposed solution.

Keywords: Internet Portal, Architecture Design, PHP, Nette

Jsem velmi rád, že na tomto místě mohu poděkovat vedoucímu své práce, panu Ing. Radku Valovi, Ph.D., za obětavost, připomínky a cenné rady při vedení mé diplomové práce. Dále bych chtěl poděkovat své rodině za pomoc a trpělivost.

## OBSAH

ÚVOD .....	10
<b>I TEORETICKÁ ČÁST</b> .....	<b>10</b>
<b>1 NÁSTROJE A SLUŽBY ZAJIŠŤUJÍCÍ POTŘEBY PROJEKTU</b> .....	<b>12</b>
1.1 COMPOSER.....	12
1.2 REDIS .....	13
1.3 GIT .....	14
1.4 GITHUB.....	14
1.5 BITBUCKET.....	15
1.6 DEPLOY .....	15
<b>2 FRAMEWORK NETTE</b> .....	<b>17</b>
2.1 STRUKTURA FRAMEWORKU .....	17
2.2 KONFIGURACE FRAMEWORKU A DEPENDENCY INJECTION .....	17
2.3 ŽIVOTNÍ CYKLUS PRESENTERU .....	18
2.4 SYSTÉM ŘÍZENÍ UŽIVATELSKÝCH OPRÁVNĚNÍ .....	20
2.5 CACHE.....	20
2.6 DATABASE.....	22
2.6.1 Vrstva Table .....	23
<b>3 DOPLŇKY PRO FRAMEWORK NETTE</b> .....	<b>25</b>
3.1 KDYBY/REDIS.....	25
3.2 KDYBY/EVENTS .....	26
3.3 O5/GRIDO .....	27
<b>4 SLEDOVANÉ PARAMETRY VÝKONU SERVERU</b> .....	<b>29</b>
4.1 CACHE HIT RATIO .....	29
4.2 LOAD AVERAGE .....	29
<b>II PRAKTICKÁ ČÁST</b> .....	<b>30</b>
<b>5 ZADÁNÍ</b> .....	<b>32</b>
5.1 KLIENSKÉ POŽADAVKY.....	32
5.1.1 Tématické členění obsahu .....	32
5.1.2 Strukturované URL .....	32
5.1.3 Systém řízení uživatelských oprávnění.....	32
5.2 FIREMNÍ POŽADAVKY.....	33
5.2.1 Framework Nette na platformě PHP .....	33

5.2.2	Automatizovaný proces aktualizace.....	33
5.2.3	Dynamické formuláře .....	33
5.2.4	Znovupoužitelné komponenty uživatelského rozhraní.....	34
5.2.5	Datové gridy s možností inline editace .....	34
5.2.6	Flexibilní datová vrstva.....	34
5.2.7	Databázová vrstva Database.....	34
5.2.8	Využití cache .....	34
<b>6</b>	<b>NÁVRH ŘEŠENÍ.....</b>	<b>35</b>
6.1	FRAMEWORK NETTE NA PLATFORMĚ PHP.....	35
6.2	AUTOMATICKÝ PROCES AKTUALIZACE.....	36
6.2.1	Aktualizace struktury databáze .....	36
6.3	TÉMATICKÉ ROZDĚLENÍ OBSAHU.....	39
6.4	STRUKTUROVANÉ URL.....	41
6.5	DATABÁZOVÁ VRSTVA NETTE/DATABASE.....	43
6.6	DYNAMICKÉ FORMULÁŘE .....	46
6.7	FLEXIBILNÍ MODELOVÁ VRSTVA .....	50
6.8	ZNOVUPOUŽITELNÉ KOMPONENTY UŽIVATELSKÉHO ROZHRANÍ .....	51
6.9	DATOVÉ GRIDY S MOŽNOSTÍ INLINE EDITACE.....	52
6.10	SYSTÉM ŘÍZENÍ UŽIVATELSKÝCH OPRÁVNĚNÍ .....	53
6.10.1	System pro převod vnitřní adresy na parametry ACL .....	55
6.11	VYUŽITÍ CACHE.....	59
<b>7</b>	<b>IMPLEMENTACE.....</b>	<b>62</b>
7.1	FRAMEWORK NETTE NA PLATFORMĚ PHP.....	62
7.1.1	Režim maintenance.....	62
7.1.2	Vymazání cache .....	63
7.1.3	Vymazání obsahu adresáře temp .....	63
7.1.4	Implementace nástroje pro aktualizaci databáze.....	64
7.2	TÉMATICKÉ ROZDĚLENÍ OBSAHU.....	65
7.3	STRUKTUROVANÉ URL.....	65
7.4	DATABÁZOVÁ VRSTVA NETTE/DATABASE.....	68
7.5	DYNAMICKÉ FORMULÁŘE .....	73
7.6	FLEXIBILNÍ MODELOVÁ VRSTVA .....	74
7.7	ZNOVUPOUŽITELNÉ KOMPONENTY UŽIVATELSKÉHO ROZHRANÍ .....	75
7.8	INLINE EDITACE.....	77

7.9	SYSTÉM ŘÍZENÍ UŽIVATELSKÝCH OPRÁVNĚNÍ .....	77
7.10	VYUŽITÍ CACHE.....	78
<b>8</b>	<b>ZHODNOCENÍ PROVOZU .....</b>	<b>80</b>
8.1	UDRŽOVATELNOST APLIKACE.....	80
8.2	AUTOMATIZOVANÝ PROCES AKTUALIZACE .....	80
8.3	ZÁTĚŽ SERVERU.....	81
8.4	CACHE POMOCÍ SERVERU REDIS.....	81
8.5	KRITICKÉ CHYBY .....	82
	<b>ZÁVĚR.....</b>	<b>84</b>
	<b>SEZNAM POUŽITÉ LITERATURY .....</b>	<b>86</b>
	<b>SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK .....</b>	<b>89</b>
	<b>SEZNAM OBRÁZKŮ .....</b>	<b>90</b>
	<b>SEZNAM PŘÍLOH.....</b>	<b>91</b>

## ÚVOD

Návrh architektury software je proces spočívající v identifikaci podsystémů, stanovení rámce pro komunikaci a možnostech řízení. Návrhem architektury začíná proces návrhu software a tvoří tak spojnicí mezi specifikací a návrhem. Proces návrhu tedy zahrnuje identifikaci hlavních částí software a jejich propojení.

V průběhu návrhu bereme v úvahu celou řadu skutečností. Uvažujeme zda bude výsledný systém distribuovaný a zda je možné využít generickou aplikační architekturu. Dále uvažujeme nejvýhodnější architektonický styl, způsob řízení a v neposlední řadě také způsob vedení dokumentace. Velkou pozornost věnujeme stanovení procesu ověření výsledného návrhu. Cílem návrhu je především zajistit u realizovaného software výkonnost, zabezpečení, bezpečnost, dostupnost a udržitelnost.

V průběhu realizace projektu tvoří návrh architektury významnou část a na jeho kvalitě do značné míry závisí úspěšnost výsledné realizace. Návrhu je tedy nutné věnovat maximální pozornost a pečlivě zvážit všechna rozhodnutí.

Při procesu návrhu architektury je však nutné respektovat možnosti, schopnosti a zvyklosti vývojového týmu. Některé technologie či postupy jsou považovány firmou, potažno vývojovým týmem, pro konkrétní účel použití jako obvyklé. Jejich nahrazení jinými technologiemi, byť pro daný účel vhodnějšími, by mohlo způsobit nemalé obtíže. Z těchto důvodů jsou pro některé úlohy využity konkrétní technologie bez diskuze vhodnosti jejich použití.

Použití aplikačního frameworku přispívá k celkové úspěšnosti realizace projektu. Framework nabízí nástroje pro realizaci funkcionalit, které se obvykle vyskytují u většiny moderního software. Jejich využití dovolí soustředit se pouze na funkce realizovaného business procesu. Možnosti frameworku jsou rozšiřovány množstvím dostupných doplňků. Doplňky ve formě knihoven či celých modulů usnadňují řešení běžných funkcionalit a také samy mnohdy realizují komplexní úlohy. Použití doplňků v aplikaci tak přispívá ke snižování nákladů na vývoj.

Ve své práci se zaměřuji na návrh architektury internetového portálu fotbal.cz. Ve fázi specifikace zadání jsou stanoveny klíčové požadavky kladené na architekturu aplikace. Tyto požadavky jsou postupně zpracovávány v kapitolách návrhu řešení. Nejprve je navržen způsob řešení a poté jsou zvoleny technologie, které budou pro realizaci použity. V kapitole věnované popisu implementace je uveden průběh a výsledek finální implementace uskutečněné dle připraveného návrhu. Jsou zde popsány klíčové momenty implementace a možnosti konfigurace realizovaných modulů. V závěru práce je provedeno zhodnocení provozu portálu po spuštění do produkčního režimu.

# I. TEORETICKÁ ČÁST

## 1 NÁSTROJE A SLUŽBY ZAJIŠŤUJÍCÍ POTŘEBY PROJEKTU

Tato kapitola je věnována popisu nástrojů a služeb, které jsou použity při realizaci projektu a také pro zabezpečení plynulého vývoje a provozu.

### 1.1 Composer

Composer je svobodný nástroj pro správu závislostí PHP aplikací. Umožňuje určit seznam balíčků, které projekt využívá. Následně je pro tento seznam řízen proces instalace a aktualizace se zajištěním kompatibility verzí.

Balíček je v terminologii Composeru označení pro modul, knihovnu či celý projekt. Balíček představuje samostatnou jednotku a je definován svým globálně unikátním názvem. Název balíčku je složen ze dvou částí. Části jsou od sebe odděleny znakem `/`. První část názvu tvoří jméno poskytovatele balíčku. Obvykle se používá jméno autora nebo název firmy. Druhou část tvoří vlastní název balíčku. Tento název by měl co nejvíce vystihnout obsah či účel balíčku. [1]

Composer využívá při instalaci jako zdroj dat uložště balíčků. Uložště může být definováno více způsoby. Centrální uložště balíčků představuje systém Packagist, který je hostován na adrese <https://packagist.org/>. Alternativně může být uložště balíčků tvořeno vlastním privátním systémem. Systém může být lokální či distribuovaný. [2]

Konfigurace nástroje Composer je tvořena jedním strukturovaným souborem `composer.json` ve formátu JSON. Konfigurační soubor je rozdělen do více sekcí. Nalezneme zde sekce pro nastavení základních informací o balíčku, sekce pro definici závislostí, sekce pro nastavení autoloaderu a sekce pro definici dodatečných datových uložšť.

Typický konfigurační soubor vypadá následovně. [3]

```
{
  "name": "monolog/monolog",
  "type": "library",
  "description": "Logging for PHP 5.3",
  "keywords": ["log", "logging"],
  "homepage": "https://github.com/Seldaek/monolog",
  "license": "MIT",
  "authors": [
    {
      "name": "Jordi Boggiano",
      "email": "j.boggiano@seld.be",
      "homepage": "http://seld.be",
      "role": "Developer"
    }
  ]
}
```

```
    ],
    "require": {
        "php": ">=5.3.0"
    },
    "autoload": {
        "psr-0": {
            "Monolog": "src"
        }
    }
}
```

Centrální uložení balíčků Packagist spolupracuje se systémy pro hostování projektů GitHub a Bitbucket. Při změně ve zdrojových kódech v těchto systémech dochází k předání události do uložení Packagist. Packagist následně ve své databázi aktualizuje informace o aktuální verzi balíčku. V systému Packagist tak budou vždy informace o balíčku aktuální a není nutné provádět ruční synchronizaci. [4]

Pomocí nástroje Composer lze efektivně řešit situaci, kdy z důvodů rozšíření API provedeme klonování určitého balíčku. Balíček bude následně doplněn o vlastní funkce a bude opatřen novým názvem. Při řešení závislostí mezi balíčky by ovšem došlo ke kolizní situaci. Některé balíčky by mohly vyžadovat přítomnost zdrojové knihovny. Composer by při řešení závislostí stáhnul jak původní verzi knihovny, tak i naši novou. Aby k této situaci nemohlo dojít, lze využít v konfiguračním souboru nové knihovny sekci *replace*. Do této sekce se uvede jméno a verze zdrojové knihovny. Composer poté při řešení závislostí považuje novou knihovnu zároveň za původní. [3]

## 1.2 Redis

Redis je svobodné *in-memory* uložení datových struktur. Jedná se o *key-value* databázi, kde pro jeden klíč lze uložit jednu nebo i více datových struktur. Podporovány jsou různé datové struktury jako string, hash, list, set, bitmap, geindex a další. Redis lze využít jako NoSQL databáze, cache server nebo jako Message Broker. Podporovány jsou také síťové operace a schopnost běhu v roli síťového serveru. Síťové operace jsou využity i pro komunikaci mezi klientem a serverem pomocí protokolu TCP. [5][6]

Redis primárně data uchovává v operační paměti. Ukládání obsahu paměti na disk je řízeno nastavením konfiguračního souboru. Pokud je Redis využit jako uložení pro paměť cache, nejsou data na disk ukládána nikdy. Běh Redisu není omezen velikostí dostupné operační paměti. Lze využít i odkládací prostor. [6]

Podpora škálovatelnosti je u Redisu zajištěna pomocí replikace. Redis podporuje replikaci typu MASTER-SLAVE. Všechny instance SLAVE umožňují pouze čtení. Zapisovat lze pouze v instanci MASTER. [5]

Redis je díky využití operační paměti velmi výkonný. Dokáže zajistit velmi vysoký počet operací. Disponuje nástrojem benchmark. Benchmark je určeným pro testování výkonu a lze tak zjistit aktuální propustnost a rychlost odezvy. [7]

### 1.3 Git

Git je svobodný, distribuovaný systém správy verzí. Původně Git sloužil pro vývoj jádra Linuxu, později se však rozrostl na kompletní systém správy revizí. [8] Charakteristické vlastnosti jsou: [8] [9]

- Dostupnost ve většině hlavních operačních systémů.
- Distribuovaný vývoj.
- Podpora pro nelineární vývoj. Je podporováno rychlé větvení a slučování.
- Podpora protokolů SSH, HTTP, FTP, rsync.
- Rychlost, škálovatelnost.
- Zaměnitelné slučovací strategie.
- Implicitní přejmenování souborů.

### 1.4 GitHub

GitHub je webově orientovaná služba pro poskytování hostingu projektům verzovaných pomocí systému správy revizí Git. Poskytuje služby distribuované správy revizí a správy zdrojových souborů. Nabízí podporu pro služby řízení přístupu, kde lze jednotlivým repozitářům nastavovat různé úrovně oprávnění. Disponuje přehledným webovým grafickým prostředím zajišťujícím potřeby všech prováděných operací. Github je zaměřen na podporu týmového vývoje. Nabízí širokou paletu služeb usnadňující práci v týmu. [10]

GitHub podporuje svou cenovou politikou open-source projekty. Založení a udržování veřejně dostupného repozitáře není zpoplatněno a služby GitHubu jsou zdarma. Pro soukromé repozitáře je zapotřebí vybrat jednu z placených nabídek. [11]

Předností GitHubu je snadná propojitelnost s uložištěm balíčků Packagist a mnoha dalšími systémy. [4]

## 1.5 Bitbucket

Bitbucket je obdobně jako GitHub webově orientovaná služba pro poskytování hostingu projektům verzovaných pomocí systému správy revizí Git a Mercurial. Poskytuje distribuovanou správu revizí, disponuje webovým grafickým rozhraním a také poskytuje služby pro řízení přístupu k repositáři. Bitbucket podporuje tvorbu pull-requestů a řízení přístupu uživatelů k jednotlivým větvím repositáře. Stejně jako GitHub podporuje práci v týmu. [12][13]

Na rozdíl od GitHubu však Bitbucket poskytuje své služby zdarma týmu čítajícímu maximálně pět členů. V rámci tohoto účtu lze vytvářet jak volně přístupné repositáře, tak i repositáře soukromé. Služby pro větší týmy jsou placené. [14]

## 1.6 Deploy

Deploy je služba zajišťující proces distribuce aplikace na produkční server. Samotný proces distribuce je složen z více typů prováděných operací. Operace mohou být například kopírování zdrojových kódů aplikace, zapnutí režimu maintenance na cílovém serveru, provedení změn v relační databázi, vymazání obsahu paměti cache a spuštění skriptů automatického testování.

Deploy vyžaduje specifikovat zdrojový Git repositář, který je považován za primární vstup. Deploy poté hlídá změny v primárním repositáři. Zjištěný rozdíl je pak považován za změnu, kterou je nutné propagovat na cílový server. Primární repositář musí být umístěn v systém Codebase, GitHub nebo Bitbucket, se kterými je Deploy plně integrován. Služba Deploy podporuje přístup na cílový server pomocí protokolu SSH. Na cílovém serveru tak lze spouštět potřebné shellové skripty. [15]

Služby poskytované Deployem jsou pro jeden projekt s omezením na maximálně 10 aktualizací denně zdarma. Pro více projektů nebo množství aktualizací je zapotřebí využít některého z placených tarifů. [16]

Charakteristické vlastnosti služby Deploy jsou: [15]

- On-line monitorování procesu aktualizace prostřednictvím webového UI.
- Automatické spuštění procesu aktualizace po provedení změny ve sledovaném repositáři.
- Podpora příkazů spouštěných na cílovém serveru (pouze pro SSH přístup).
- Centrální správa konfiguračních souborů, které mohou být distribuovány na cílový server.

- Snadná integrace s mnoha hostingovými servery pro repozitáře jako je GitHub, Bitbucket a další.
- Podpora více módu aktualizace (vývoj, test, produkce).
- Řízení přístupu uživatelů k projektům.

## 2 FRAMEWORK NETTE

Nette je framework určený primárně pro vývoj webových aplikací v jazyce PHP. Za jeho vznikem a neustálým vývojem stojí David Grudl. Za rozvojem Nette také stojí organizace Nette Foundation. [17] Do Nette ovšem přispívá i řada jiných vývojářů. Aktuální verze v době vzniku této práce je 2.3. Od verze 2.2 není framework koncipován jako monolit, ale je rozdělen do samostatných balíčků. Lze tak ve vyvíjené aplikaci využít pouze ty části frameworku, které opravdu potřebujeme. [18]

Dokumentace je umístěna na oficiálním webu frameworku a obsahuje základní informace a příklady použití. David Grudl mimo vývoje pořádá i školení, ve kterých se věnuje nejen základům, ale i pokročilým technikám použití frameworku. [19]

Zdrojové kódy jsou umístěny na serveru GitHub a balíčky pro Composer jsou dostupné v systému Packagist. [20]

Předností Nette je fakt, že se jedná o svobodný software, který je nabízen pod licencí GNU GPL a licencí Nette. [17]

### 2.1 Struktura frameworku

Framework Nette aktuálně nabízí celkem 22 samostatných balíčků. Jedním z balíčků je *nette/nette*, který zůstává z důvodu zpětné kompatibility. Řada rozšíření či aplikací stojících na Nette stále využívá v závislostech *nette/nette*. Jednotlivé balíčky frameworku mají vždy konkrétní použití a řeší vždy jednu oblast potřebnou pro vývoj webové aplikace. Příkladem samostatných balíčků může být šablonovací systém Latte, nástroj pro ladění Tracy, systém pro přístup k databázi Database a řada dalších. [21]

### 2.2 Konfigurace frameworku a Dependency Injection

Framework Nette je řízen konfigurací zapsanou pomocí konfiguračních souborů. Konfigurační soubory jsou zpracovávány překladačem z balíčku *nette/neon*. Formát souborů tedy odpovídá syntaxi Neon. Formát Neon je obdobný formátu YAML. Konfiguraci lze díky podpoře vkládání dalších souborů rozdělit do více samostatných souborů. Jednotlivé soubory pak obsahují nastavení některého z logických celků konfigurace. Příkladem může být soubor pro nastavení registrace rozšíření DI kontejneru, soubor pro konfiguraci databáze, soubor s nastavením parametrů jazyka PHP a další. Výsledná konfigurace je tak přehledná a názorná na pochopení.

Nette využívá techniku Dependency Injection (DI), která zajišťuje správu a předávání závislostí mezi objekty. [23] Podpora pro DI je dostupná v balíčku *nette/di*. Základ pro DI v Nette představuje třída *Container*. Prvním krokem po spuštění aplikace je

vytvoření instance třídy *Configurator*. Tato třída je zodpovědná za zpracování konfigurace a tvorbu finálního DI kontejneru. Finální DI kontejner je vytvořen na základě nastavení uvedeném v konfiguračních souborech aplikace. Je realizován vlastní třídou dědicí od *Container* a je umístěn mezi dočasnými soubory aplikace.

Konfigurační soubor definuje pravidla pro tvorbu instancí tříd a určuje předávané parametry. Mohou zde být také uvedeny rozšiřující pravidla, která jsou využita při tvorbě instancí třídy. Nette DI umí automaticky implementovat továrny pro třídy. Stačí pouze vytvořit rozhraní obsahující metodu *create* a příslušnou anotaci specifikující návratový typ. Implementovaným továrnám lze samozřejmě v konfiguraci řídit předávané parametry. [22][24]

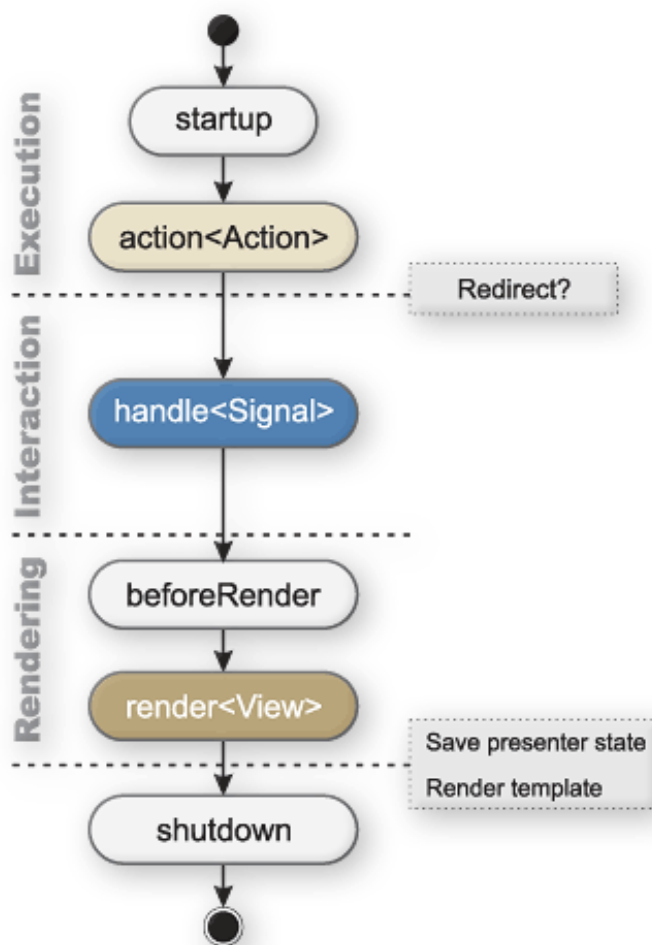
### 2.3 Životní cyklus presenteru

Znalost životního cyklu presenteru je vyžadována pro efektivní tvorbu aplikace ve frameworku Nette. Životní cyklus presenteru je implementován abstraktní třídou *Presenter*. Tato třída je použita jako předek pro všechny vytvářené aplikační presentery. *Presenter* implementuje rozhraní *IPresenter*, které definuje vstupní bod aplikace. Tento vstupní bod vyžaduje třída *Application*, která zodpovídá za životní cyklus celé aplikace. Třída *Presenter* a další třídy realizující proces zpracování požadavků jsou obsaženy v balíčku *nette/application*.

Životní cyklus presenteru se skládá celkem z šesti kroků. Každému kroku odpovídá jedna speciální metoda. Názvy metod se tvoří podle pravidla camel case. [25]

Životní cyklus je prováděn v krocích: [25]

- První krok představuje volání metody *startup*. Metoda *startup* slouží k provádění činností, které se mají vždy vykonat bez ohledu na cílovou akci. Typicky se zde provádí například autorizace uživatele.
- Druhý krok představuje volání metody *action*. Název metody je tvořen prefixem *action* a suffixem odpovídající názvu požadované akce. Například pro akci *default* se bude metoda jmenovat *actionDefault*. Metody *action* slouží především k ověření vstupních parametrů či vykonáním požadovaných příkazů. V metodě *action* může být změněna metoda pro vykreslení. Za normálních okolností bude volána metoda vykreslení *render* odpovídající názvu akce. Pro akci *default* to bude *renderDefault*. Pomocí volání *Presenter::setView*, může být metoda pro vykreslení změněna. Metoda *action* patří mezi nepovinné. Pokud není definována, realizuje se další krok.
- Třetí krok životního cyklu odpovídá volání metody *handle* pro zpracování signálu. Tato metoda se volá pouze tehdy, je-li signál definován. Pokud není zadán signál,



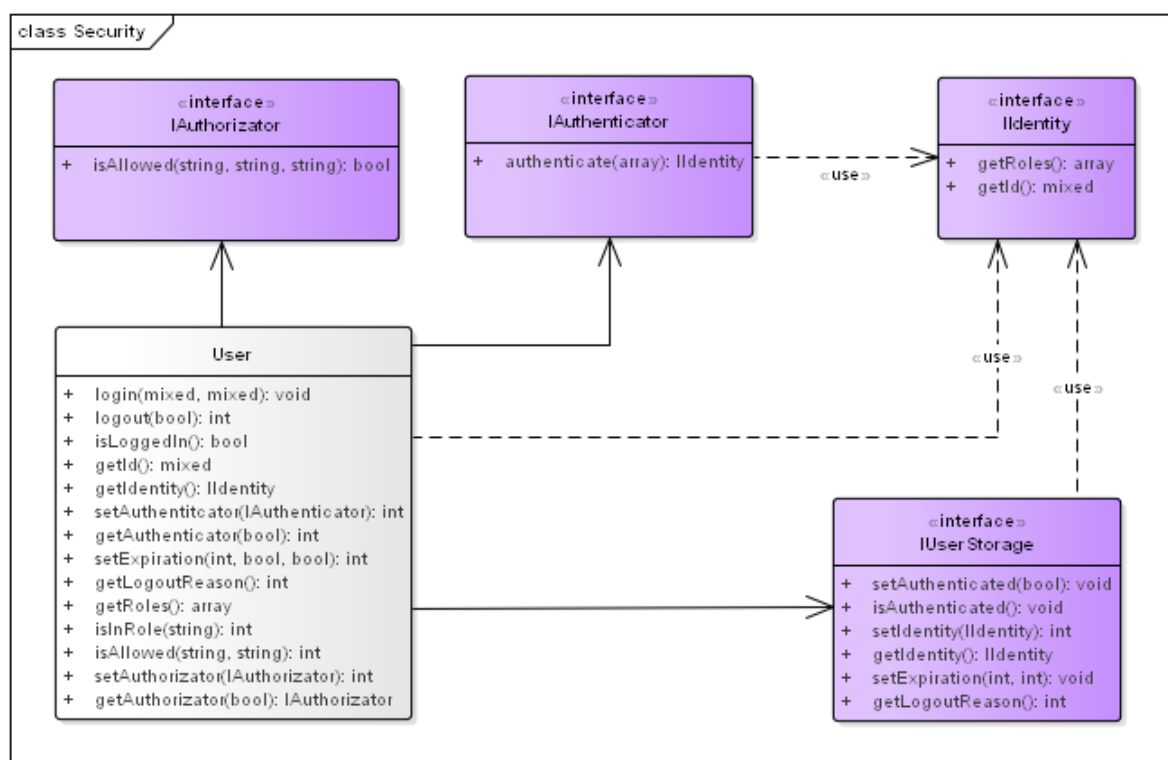
Obr. 2.1 Životní cyklus presenteru[25]

tato metoda se přeskakuje. Název metody *handle* je opět složen z prefixu *handle* a suffixu odpovídající názvu zpracovávaného signálu.

- Čtvrtý krok tvoří volání metody *beforeRender*. Tato metoda se volá vždy bez ohledu na zvolenou akci. Typickým uplatněním může být naplnění šablony informacemi, které jsou v šabloně zapotřebí bez ohledu na požadovanou akci.
- Pátý krok tvoří volání metody *render*. Metoda *render* je opět složena z prefixu *render* a suffixu odpovídajícímu názvu akce. Pro akci *default* bude příslušný název metody *renderDefault*. Tato metoda je typicky využita pro předání dat šabloně.
- Šestý krok životního cyklu je realizován voláním metody *shutdown*. Tato metoda je volána při ukončení životního cyklu presenteru. Metodu je možné využít například pro uzavření otevřeného streamu a podobně.

## 2.4 Systém řízení uživatelských oprávnění

Podporu pro řízení uživatelských oprávnění ve frameworku Nette představuje balíček *nette/security*. Balíček *nette/security* obsahuje model uživatelského účtu včetně implementace procesu autentifikace a autorizace. Model uživatelského účtu je znázorněn pomocí diagramu tříd na obrázku 2.2. Uživatelský účet je tvořen třídou *User*, rozhraním *IAuthenticator* a *IAuthorizator*. Třída *User* představuje model uživatele. Obsahuje metody pro autentifikaci, autorizaci, odhlášení a další metody řízení přístupu. Třída *User* vyžaduje pro svou činnost instanci implementující rozhraní *IUserStorage*. Rozhraní *IUserStorage* slouží k zajištění perzistence údajů o přihlášení. Uložiště může být libovolné. Podporován je například souborový systém nebo *keyvalue* server Redis. [26]



Obr. 2.2 Diagram tříd uživatelského účtu frameworku Nette

## 2.5 Cache

Nette disponuje v balíčku *nette/cache* velmi rozsáhlou podporou pro práci s pamětí cache. Problematika je podle kontextu operací rozdělena do dvou vrstev. [27]

První vrstvu tvoří specifikace uložiště dat paměti cache. Uložiště je definováno rozhraním *IStorage*. *IStorage* obsahuje definici metod pro čtení, zápis, zamykání a mazání obsahu dle kritérií.

Nette cache obsahuje v základu několik implementací rozhraní *IStorage*. Podporo-

vány jsou uložisté: [28]

- **FileStorage** je uložisté realizované pomocí souborového systému.
- **MemoryStorage** je uložisté umístěné v lokální paměti této třídy. Po ukončení životnosti objektu zaniknou i uložená data.
- **MemcachedStorage** je adaptér pro server Memcached.
- **SQLiteStorage** je adaptér pro databázi SQLite.
- **DevNullStorage** je *NullObject* implementace. Využívá se pro ladění aplikace.

Druhá vrstva je tvořena třídou *Cache*. *Cache* zajišťuje operace pro manipulaci s uložistěm paměti cache (rozhraní *IStorage*). V aplikaci by mohly nastat situace, kdy by byly vícekrát použity stejné klíče. Tato situace by mohla lehce nastat například v prezentační vrstvě. Třída *Cache* z těchto důvodů umožňuje rozdělení obsahu pomocí jmenných prostorů.

Data ukládaná do paměti cache prostřednictvím třídy *Cache* je možné doplnit o doplňkové řídicí údaje. Řídicí údaje se využívají pro řízení platnosti dat v paměti cache a k jejich označení tagy. Tagy se používají pro řízení procesu zneplatnění obsahu. V praxi totiž nastávají situace, kdy se jednoho subjektu týká více různých záznamů v paměti. Při změně subjektu chceme docílit stavu, kdy budou zneplatněna všechna data související s tímto subjektem. Tato situace je řešitelná právě použitím tagů. Oddělené paměťové místa se označí stejným tagem. Pro zneplatnění obsahu paměti pak stačí pouze zadat použitý tag.

Následující příklad nastavuje datům vkládaných do paměti maximální životnost 20 minut a označuje je dvěma tagy. [27]

```
<?php
$cache->save($articleId, $html, [
    Cache::EXPIRE => '20 minutes',
    Cache::TAGS => ["article/$articleId", "comments/$articleId"],
]);
```

Vymazání obsahu paměti cache podle tagu je uvedeno na příkladu: [27]

```
<?php
$cache->clean([
    Cache::TAGS => ["article/$articleId"],
]);
```

Použití paměti cache je podporováno i šablonovacím systémem Latte. K dispozici je zde makro *cache*. Makro *cache* zajistí uložení obsahu mezi počáteční a koncovou značkou makra. Jako identifikátor bude použit parametr uvedený v makru. Pokud identifikátor v makru chybí, makro jako identifikátor vygeneruje náhodný řetězec znaků.

Příklad použití makra *cache*: [27]

```
{cache $id, expire => '20 minutes', tags => [tag1, tag2]}
...
{/cache}
```

## 2.6 Database

Database je název pro vrstvu zajišťující přístup k relačním databázím. Implementace této vrstvy je dostupná v balíčku *nette/database*. Předností Database je snadné skládání dotazů s následným získáním a zpracováním výsledků. [29]

Pro práci s Database je nutné nejprve vytvořit instanci třídy *Connection*. *Connection* zajišťuje připojení k databázovému serveru pomocí rozhraní PDO jazyka PHP. Vytvoření instance vyžaduje specifikovat parametr DNS. Tento parametr specifikuje adresu databázového serveru, ke kterému se chceme připojit. Dalšími parametry jsou autentifikační údaje a konfigurace připojení. Konfigurací připojení lze nastavit databázový ovladač či vlastnost *lazy*. Pokud je *lazy* povoleno, připojení k databázovému serveru se uskuteční teprve při prvním požadavku do databáze. Parametry konfigurace se také předávají použitému databázovému ovladači. Lze tedy současně provést nastavení vrstvy PDO a databázového ovladače. [29] [30]

Database aktuálně podporuje tyto relační databáze. [29]

Databázový server	DSN jméno	Podpora v Database	Podpora v Table
MySQL	mysql	ANO	ANO
PostgreSQL	pgsql	ANO	ANO
Sqlite 3	sqlite	ANO	ANO
Sqlite 2	sqlite2	ANO	–
Oracle	oci	ANO	–
MS SQL (PDO_SQLSRV)	sqlsrv	ANO	ANO
MS SQL (PDO_DBLIB)	mssql	ANO	–
ODBC	odbc	ANO	–

Pokládání dotazů do databáze se realizuje pomocí třídy *Context*, konkrétně pomocí metody *Context::query*.

```
<?php
use Nette\Database\Context;
```

```
$database = new Context($connection);

$database->query('INSERT INTO users', array( // parametr může být pole
    'name' => 'Jim',
    'created' => new DateTime, // nebo objekt DateTime
    'avatar' => fopen('image.gif', 'r'), // nebo soubor
), ...); // je možné také provést více vložení najednou

$database->query('UPDATE users SET ? WHERE id=?', $data, $id);
$database->query('SELECT * FROM categories WHERE id=?', 123);
```

### 2.6.1 Vrstva Table

Table je označení podvrstvy, která je součástí Database. Je navržena pro zjednodušený a optimalizovaný výběr dat z databáze. Hlavní myšlenkou Table je neskládat více SQL dotazů pro výběr dat, ale načítat obsah jedné tabulky tak, aby se tyto dotazy pokládaly jen jednou. [29]

Table implementuje Table Data Gateway a Row Data Gateway pattern. Table Data Gateway je objekt, který slouží jako brána do databázové tabulky. Row Data Gateway je objekt, který reprezentuje jeden záznam z datového zdroje. [31]

Databázová tabulka je ve vrstvě Table reprezentovaná třídou *Selection*. Instanci *Selection* lze získat z voláním *Context::table*. Třída *Selection* obsahuje metody umožňující zadávat kritéria projekce i selekce. Třída *Selection* implementuje rozhraní *Traversable* jazyka PHP, které umožní iterovat přes objekty *Selection*. Iterací je vrácen obsah řádku v podobě instance třídy *ActiveRow*. [29]

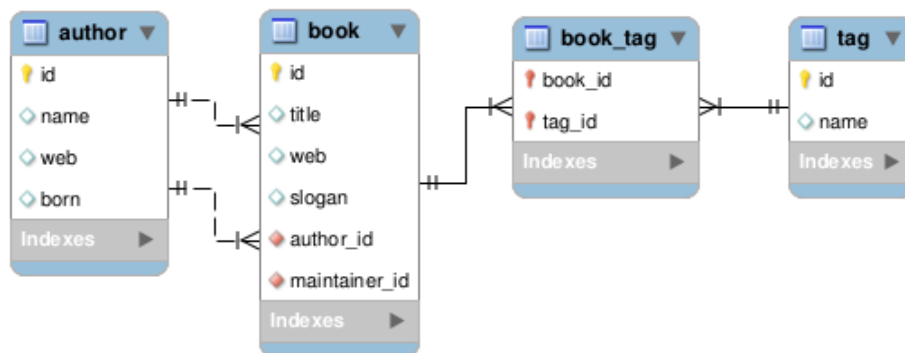
Table spravuje vazby mezi tabulkami dvěma způsoby: [29]

- Filtrace záznamů vracející instanci třídy *Selection*
- Získání souvisejících dat vybraných instancí třídy *ActiveRow*

V následujícím příkladu je vysvětleno chování vztahů mezi tabulkami při použití Table. Schéma databáze je uvedeno na obrázku 2.3.

Příklad znázorňuje získání souvisejících dat pro vybrané knihy. Proměnná *author* řádku tabulky *book* reprezentovaná objektem *ActiveRow* obsahuje další objekty *ActiveRow* reprezentující autora knihy. Tagy knihy jsou získány metodou *Selection::related*. Návrátovou hodnotou je příslušná kolekce. V cyklu je získáno jméno tagu. Jméno je uloženo v proměnné *tag* objektu *ActiveRow* právě procházené vazby. [29]

```
<?php
$books = $context->table('book');
```



Obr. 2.3 ER diagram databáze k příkladu

```
foreach ($books as $book) {  
    echo 'title:      ' . $book->title;  
    echo 'written by: ' . $book->author->name;  
  
    echo 'tags: ';  
    foreach ($book->related('book_tag') as $bookTag) {  
        echo $bookTag->tag->name . ', ';  
    }  
}
```

### 3 DOPLŇKY PRO FRAMEWORK NETTE

Pro framework Nette existuje nepřeberné množství různých doplňků. Tyto doplňky realizují nejen nejčastější funkce využívané webovými aplikacemi, ale i adaptéry různých služeb včetně komponent dalších frameworků. Množství dostupných doplňků mnohokrát znásobuje použitelnost samotného frameworku, neboť se vývojář může soustředit na činnost realizovaného produktu.

#### 3.1 kdyby/redis

Balíček *kdyby/redis* poskytuje adaptéry umožňující využít ve frameworku Nette služeb serveru Redis. Server Redis může být v aplikaci nasazen jako úložiště pro: [32]

- Hlavní úložiště obsahu paměti cache. Balíček disponuje implementací rozhraní `IStorage` a může být nasazen místo výchozího souborového úložiště. Použitím serveru Redis jako hlavní úložiště je docíleno velkého cache, neboť server Redis je navržen právě pro operace velmi rychlého přístupu k datům.
- Úložiště žurnálu systému řízení paměti cache. Nette využívá ve výchozím stavu souborový žurnál, ve kterém udržuje informace potřebné pro řízení expirace a zneplatnění dat. Žurnál představuje u velké aplikace úzké hrdlo z důvodu velkého počtu operací. Využitím serveru Redis jako úložiště žurnálu dochází ke stabilizaci výkonu.
- Ukládání záznamů session. Výchozím nastavením úložiště session bývá u většiny serverů souborový systém. Aplikace spravující stovky či tisíce záznamů může citelně omezit výkon aplikace, neboť nastartování session může být opožděno právě v důsledku velkého množství záznamů. Využitím serveru Redis lze problém elegantně předejít.

Balíček obsahuje pro zajištění snadné integrace do aplikace rozšíření pro DI kontejner. Konfigurace je tak velmi jednoduchá a přehledná. Příkladem výchozí konfigurace může být toto nastavení: [32]

```
extensions:  
    redis: Kdyby\Redis\DI\RedisExtension  
  
redis:  
    journal: on  
    storage: on  
    session: on
```

### 3.2 kdyby/events

Balíček *kdyby/events* nabízí robustní systém řízení událostí. Události jsou ve frameworku Nette implementovány pomocí třídy *Object*. Všechny třídy od ní odvozené pak disponují schopností generovat události. Implementace událostí je však v této podobě velmi jednoduchá a do značné míry omezující. Nehodí se pro realizaci skutečného systému řízeného pomocí událostí. Příklad generování a zpracování události pomocí implementace Nette je uveden zde: [33]

```
<?php
class OrderProcess extends Nette\Object
{
    public $onSuccess = array();

    private $orders;

    public function __construct(Orders $orders)
    {
        $this->orders = $orders;
    }

    public function process($values)
    {
        if ($order = $this->orders->create($values)) {
            $this->onSuccess($this, $order);
        }
    }
}

$process = new OrderProcess($orders);
$process->onSuccess[] = function ($process, $order) {
    echo "You've spent ", $order->sum, ",-";
};
```

Balíček *kdyby/events* využívá implementace Observer patternu projektu Doctrine2 z balíčku *doctrine/common*. Cílem je zkombinovat realizaci událostí projektu Doctrine a frameworku Nette. Pomocí implementace z projektu Doctrine2 se realizují listenery, které naslouchají a v případě vygenerování události zajistí její zpracování. [33]

Řízení událostí balíčkem *kdyby/events* spočívá v tvorbě subscriberu. Subscriber je tvořen třídou implementující rozhraní *Subscriber*. Subscriber poskytuje mapování místa vzniku událostí na obslužnou metodu. Registrace subscriberu do systému správy událostí (event manager) lze zjednodušit použitím speciálního tagu *kdyby.subscriber*. Tento tag se uvádí při registrování listeneru v konfiguračním souboru DI kontejneru aplikace. Příklad registrace listeneru je uveden zde: [33]

```
services:
  foo:
    class: App\FooListener
    tags: [kdyby.subscriber]
```

Pro zajištění snadné integrace do aplikace disponuje balíček rozšířením pro DI kontejner. Začlenění do aplikace je provedeno v konfiguračním souboru DI kontejneru aplikace tímto nastavením:

```
extensions:
  events: Kdyby\Events\DI\EventsExtension
```

### 3.3 o5/grido

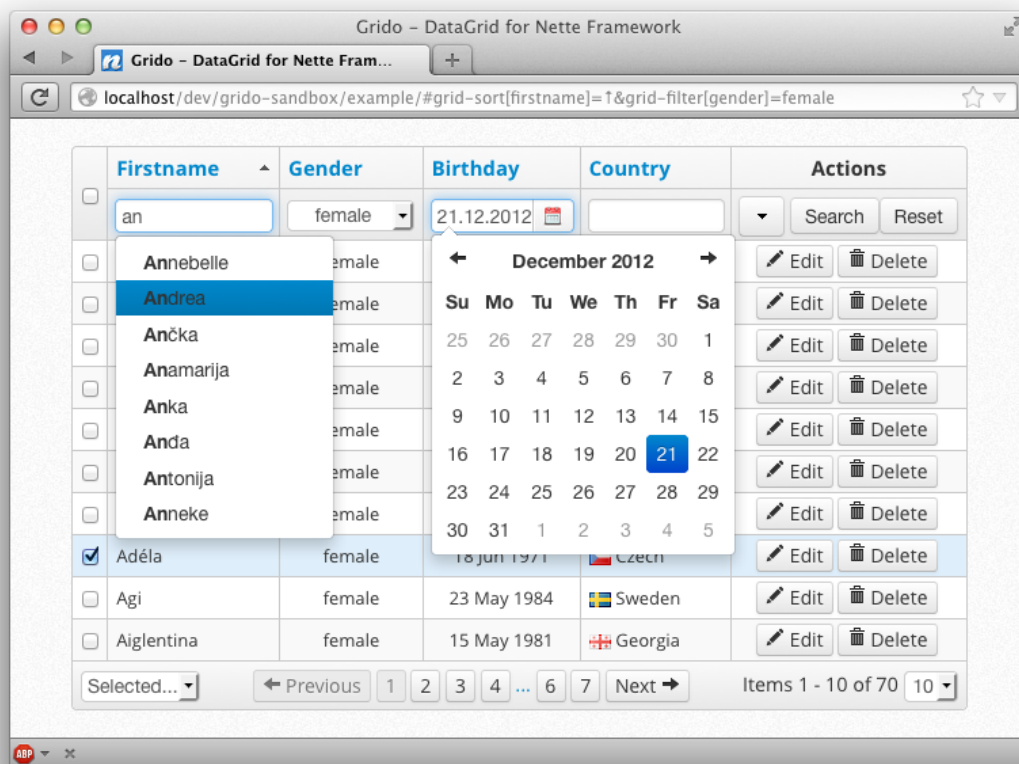
Datový grid je označení pro inteligentní tabulku uživatelského rozhraní. Jeden řádek tabulky zobrazuje jeden konkrétní záznam. Sloupce tabulky zobrazují jednotlivé atributy řádku. Výhodou tabulkového výpisu je především univerzálnost použití, přehlednost a konzistentní vzhled. Součástí datových gridů jsou zpravidla filtry. Filtry jsou určeny pro vyhledávání konkrétních záznamů podle nastavených kritérií. Datové gridy zpravidla umožňují stránkování, inline editace, formátování a další činnosti.

Všechny vyjmenované funkce realizuje komponenta Grid z balíčku *o5/grido*. Předností datového gridu realizovaného pomocí komponenty Grid je snadná použitelnost a především široká schopnost přizpůsobení. Komponenta Grid je totiž tvořena stromem komponent. Jednotlivé komponenty lze bez problémů nahrazovat jinými či je pružně rozšiřovat. Obrázek 3.1 zobrazuje příklad možného vzhledu datagridu. [34]

Data, se kterými bude Grid pracovat, je nutné popsat vhodným abstraktním modelem. Abstraktní model zaručí nezávislost na konkrétním datovém zdroji. Balíček *o5/grido* používá pro abstrakci modelu rozhraní *IDataSource*. *IDataSource* specifikuje operace pro čtení, filtrování a stránkování. Model se do komponenty Grid předává pomocí metody *Grid::setModel*.

Balíček *o5/grido* obsahuje v základu implementace *IDataSource* pro nejčastěji využívané datové zdroje. Jsou to: [36]

- ***NetteDatabase*** pro přístup k datům pomocí vrstvy Database frameworku Nette
- ***Doctrine*** pro přístup k datům pomocí ORM Doctrine2.
- ***DibiFluent*** pro přístup k datům prostřednictvím knihovny Dibi.
- ***ArraySource*** pro přístup k datům uložených v asociativním poli.



Obr. 3.1 Příklad vzhledu datového gridu balíčku o5/grido[35]

V případě využití jiného uložště lze provést vlastní implementaci rozhraní *IData-Source* a zajistit tak přístup k datům.

Komponenta Grid disponuje užitečnou vlastností spočívající ve schopnosti udržet stav filtrů a stránkování v session. Při přechodu mezi stránkami je tak zachován aktuální stav datového gridu. Například pokud datový grid obsahuje tlačítko pro editaci záznamu a bylo-li stisknuto, dojde po návratu k obnovení posledního stavu. Tato vlastnost je samořejmě nastavitelná a lze ji libovolně zapnout či vypnout. [34]

## 4 SLEDOVANÉ PARAMETRY VÝKONU SERVERU

V této kapitole jsou popsány hlavní parametry, které jsou sledovány za účelem získání informací o zátěži a výkonu serveru, na kterém je provozována realizovaná aplikace.

### 4.1 Cache hit ratio

Parametr *cache hit ratio* vyjadřuje procentuální úspěšnost vyřízení požadavků. Jedná se o podíl úspěšných požadavků ku celkovému počtu. Při dotazování se do cache můžou nastat dvě situace. První situace je označena jako *cache hit*. Vyjadřuje stav, kdy pro zadaný klíč byl nalezen v cache obsah. Druhá situace je označena jako *cache miss*. Vyjadřuje stav, kdy pro zadaný klíč nebyl v cache nalezen příslušný obsah. Hodnota *cache hit ratio* je užitečná při posouzení efektivnosti využití cache. Nízká hodnota signalizuje, že při obslužení požadavků dochází ve velké míře ke stavu *cache miss*, tedy že data nejsou v cache nalezena. V situaci, kdy je cache nasazena za účelem zvýšení výkonu systému, dochází vlivem nízkého *cache hit ratio* k opačnému účinku. Zařazení cache do řetězce zpracování požadavků představuje zbytečný krok, protože je pouze malá pravděpodobnost, že požadovaná data budou v cache nalezena. [37]

### 4.2 Load average

Parametr *load average* představuje sumu počtu aktuálně běžících procesů a počtu připravených procesů čekajících ve frontě na zpracování. Označuje se jako *run-queue length*. [38]

Význam *load average* lze nejlépe popsat pro jednojádrový procesor. Důležité hodnoty jsou: [39]

- **0.0** znamená, se není zpracováván žádný proces a žádný proces nečeká ve frontě.
- **1.0** znamená hraniční hodnotu. Všechny procesy jsou průběžně zpracovávány a žádný proces nemusí čekat ve frontě.
- **>1.0** znamená, že nejsou zpracovávány všechny připravené procesy. Některé připravené procesy musí čekat ve frontě. Hodnota 2.0 by znamenala existenci celkem dvakrát tolik připravených procesů, než může být tou dobou zpracováno.

Pro vícejádrové systémy platí, že hraniční hodnota odpovídá počtu jader. Pro čtyřjádrový procesor platí, že hodnota 4 odpovídá hodnotě 1 pro jednojádrový systém. [39]

V unixových systémech je *load average* počítán pro tři časové úseky: [40]

- 1 minuta

- 5 minut
- 15 minut

Pro jednojádrový procesor by v těchto časových úsecích hodnota *load average* znamenala: [40]

- 1.05: za poslední minutu je systém průměrně přetížen o 5%. Průměrně čeká ve frontě 0.05 procesů.
- 0.70: v posledních pěti minutách bylo CPU průměrně nezaměstnáno (idle) 30% času.
- 5.09: za posledních patnáct minut byl systém průměrně přetížen o 409%. Průměrně čeká ve frontě 4.09 procesů.

Hodnotu *load average* je vhodné posuzovat vzhledem k procentuálnímu vytížení CPU. Porovnáním lze zjistit informaci o stavu systému. Platí že: [41]

- vysoká hodnota *load average* a vysoké vytížení CPU značí, že je zapotřebí pořídit výkonnější CPU s více jádry.
- nízká hodnota *load average* a vysoké vytížení CPU značí, že je zapotřebí pořídit výkonnější CPU.
- vysoká hodnota *load average* a nízké vytížení CPU značí, že je zapotřebí pořídit více CPU jader.

Situace s nízkým vytížením CPU a vysokou hodnotou *load average* je typická pro webové servery, pro které je charakteristický paralelní provoz. Jednotlivé thready vyřizují nejčastěji pouze krátké požadavky. Mohou být také velké nároky na IO operace. Tento problém lze částečně eliminovat rychlejším diskem a využitím paměti cache. [41]

## **II. PRAKTICKÁ ČÁST**

## 5 ZADÁNÍ

Návrh architektury je klíčová etapa vývoje softwarového produktu, spočívá v identifikaci hlavních částí aplikace a stanovení komunikace mezi těmito částmi. Výsledkem návrhu je pro každou identifikovanou část specifikace použitých technologií, vlastností, architektonických vzorů, rozhraní pro komunikaci, způsobu testování a dalších vlastností, které jsou nutné pro zajištění nefunkčních požadavků aplikace.

Z množiny všech požadavků které vznikly detailním zpracováním ve fázi inženýrství požadavků a mapování business procesů, jsou vybrány ty požadavky, které mají význam v procesu návrhu architektury a které identifikují použité technologie či architektonické vzory. Pro přehlednosti jsou tyto požadavky rozčleněny do dvou kategorií. Jedná se o požadavky klientské a firemní. Klientské požadavky zahrnují požadavky vyplývající ze zadání zadavatele. Firemní požadavky vzešly z firemního softwarového procesu a odráží zvyklosti a postupy aplikované při vývoji. Značná část firemních požadavků pochází z analogických požadavků obdobných, dříve realizovaných projektů.

### 5.1 Klientské požadavky

Klientské požadavky zde zahrnují požadavky vyplývající ze zadání zadavatele. Neobsahují žádné omezení co do výběru konkrétních použitých technologií.

#### 5.1.1 Tématické členění obsahu

Uživatelské rozhraní portálu bude rozděleno do samostatných sekcí. Každá sekce zobrazuje svůj tématicky specifický obsah. Při startu projektu jsou uvažovány sekce pro reprezentaci mužů, reprezentaci žen, domácí soutěže, a sekce pro FAČR. Do budoucna je plánováno rozšiřování obsahu sekcí respektive přidávání nových.

#### 5.1.2 Strukturované URL

Jednotlivé sekce portálu budou disponovat vlastní doménou třetího řádu. URL všech stránek v jednotlivých sekcích by mělo vhodným způsobem odrážet kontext zobrazovaného obsahu a mělo by být vhodným způsobem strukturováno.

#### 5.1.3 Systém řízení uživatelských oprávnění

Administraci obsahu portálu bude využívat velké množství osob. Je požadováno, aby každé osobě bylo možné přiřadit různý rozsah oprávnění a zároveň šlo pro každou osobu definovat seznam subjektů FAČR, jehož obsah může spravovat. Není požadováno, aby

seznam oprávnění osoby byl zvlášť vztažen k jednotlivým subjektům, které může tato osoba spravovat.

## 5.2 Firemní požadavky

Firemní požadavky jsou požadavky, které vzešly z firemního softwarového procesu a odrážejí zvyklosti a postupy aplikované při vývoji. Značná část požadavků určujících využití konkrétní použité technologie jsou důsledkem firemní strategie využití stejných technologií pro konkrétní typy aplikací. Cílem omezení využitých technologií je snahou o zajištění udržitelnosti systému vzhledem k znalostem a zkušenostem vývojového týmu.

### 5.2.1 Framework Nette na platformě PHP

Projekt bude realizován za pomoci jazyka PHP ve verzi 5.4 a frameworku Nette 2.3. Vhodným způsobem musí být zajištěna snadná aktualizace frameworku i dalších knihoven použitých při realizaci projektu.

### 5.2.2 Automatizovaný proces aktualizace

Model vývoje aplikace počítá s celkem třemi servery na kterých poběží tři verze aplikace ve významu vývojového serveru, testovacího serveru a produkčního serveru.

Aplikace musí být schopna automatizovaného procesu aktualizace všech tří serveru pomocí služby třetí strany Deploy. Proces aktualizace bude zajišťovat mimo aktualizace zdrojových kódů aplikace i aktualizaci struktury databáze. V průběhu aktualizace musí být zajištěno informování uživatelů portálu vhodným hlášením o dočasné nedostupnosti portálu z důvodu aktualizace.

### 5.2.3 Dynamické formuláře

Veškeré formuláře musí být schopny tvorby dynamických prvků. Jedním z prvků je myšlen dynamický kontejner obsahující různé formulářové prvky. Tento kontejner musí být schopen se ve formuláři libovolněkrát zopakovat. Druhý dynamický prvek je kontejner, který je schopen měnit složení svých prvků na základě změny hodnoty libovolného prvku formuláře. Další typ prvku tvoří podporu pro JavaScriptovou knihovnu Select2, která umožňuje vytvořit editovatelný seznam prvků.

#### 5.2.4 Znovupoužitelné komponenty uživatelského rozhraní

Portál bude po vizuální stránce obsahovat skupiny prvků, které se budou v rámci různých sekcí i v rámci jedné stránky opakovat. Je požadováno pro tyto části použít vhodný postup tak, aby se programový kód těchto prvků neopakoval. Musí být dosaženo stavu, kdy změna programového kódu této jedné části zajistí změnu chování napříč celou aplikací.

#### 5.2.5 Datové gridy s možností inline editace

Administrace aplikace bude schopna vypisovat přehled jednotlivých entit v datových gridech. Datové gridy musí umožňovat řazení, filtrování a ve speciálních případech i inline editaci obsahu zobrazované položky. Inline editace musí být vytvořena tak, aby datový grid přímo zobrazil příslušný formulářový prvek. Pro vyvolání inline editace tak nebude nutné provést speciální postup jako je například dvojklik či jiná kombinace kláves.

#### 5.2.6 Flexibilní datová vrstva

Datová vrstva aplikace musí být navržena především s ohledem na snadnou modifikovatelnost. V průběhu prací i v následném provozu bude docházet ke zpětně nekompatibilním změnám ve struktuře databáze. Vhodným způsobem musí být zajištěn stav, kdy se některé položky obsahu nesmí za žádných okolností zobrazit na frontendu aplikace. Datová vrstva musí být také navržena s ohledem na snadnou čitelnost a ovladatelnost.

#### 5.2.7 Databázová vrstva Database

Perzistence dat bude řešena pomocí databázového serveru MySQL verze 5.6 a vyšší. Pro přístup k databázovému serveru bude využita vrstva Database knihovny *nette/database*.

#### 5.2.8 Využití cache

Frontend aplikace musí vhodným způsobem implementovat paměť cache. Portál očekává vysokou návštěvnost, která bude s postupem času narůstat. Obsah paměti cache musí být vhodným způsobem při změně původních dat zneplatněn. U klíčových údajů jako jsou aktuálně hrané zápasy, musí proběhnout zneplatnění bezprostředně po změně. Jako datové uložení paměti cache bude využit server Redis.

## 6 NÁVRH ŘEŠENÍ

Požadavky popsané v předchozí kapitole identifikují hlavní části aplikace z hlediska návrhu architektury. V této kapitole bude provedeno nalezení vhodného způsobu realizace jednotlivých požadavků. Některé požadavky mají, co do výběru technologií či způsobu realizace, limitující charakter. Tyto požadavky mohou ve svém důsledku znamenat využití technologie, která není z pohledu celého systému nejvhodnější. Těmto požadavkům musí být věnována zvláštní pozornost tak, aby bylo dosaženo požadovaných vlastností s co nejmenším rizikem selhání.

Některé požadavky je možné realizovat generickým produktem. U požadavků které není možné takto realizovat, bude proveden návrh struktury a návrh rozhraní pro komunikaci. Pro důležité funkce bude stanoven způsob testování.

### 6.1 Framework Nette na platformě PHP

Verze 5.4 jazyka PHP uvedená v požadavcích má z hlediska vývoje omezující charakter. Vzhledem k nasazení aplikace na produkční server má však své opodstatnění. Verze 5.4 je běžně dostupná na všech serverových linuxových distribucích. Omezení plynoucí z této verze by se mohla projevit v průběhu času při aktualizaci knihoven třetích stran. Je pravděpodobné, že nové verze budou využívat nových vlastností jazyka PHP a budou tedy potřebovat ke svému běhu vyšší verzi jazyka. Vhodným řízením procesu aktualizace bude možné tomuto stavu předejít. S postupem času se minimální uvažovaná verze jazyka v aplikaci zvýší, neboť bude zaručena dostupnost vyšších verzí jazyka PHP na serverových distribucích.

Verze frameworku Nette bude uvažována vždy ve své poslední verzi řady 2.3. Aktuální verze frameworku zaručí opravu chyb či zvýšení výkonu v důsledku optimalizací prováděných vývojovým týmem frameworku.

Správu všech externích knihoven (včetně frameworku Nette) bude zajišťovat balíčkovací systém Composer. Composer dovoluje využít i vlastní privátní uložení místo veřejně dostupného. Forma disponuje privátním uložení, které obsahuje řadu užitečných knihoven, které tak mohou být při realizaci projektu využity. Privátní uložení je zvoleno z důvodu ochrany firemního know-how.

Vhodnou konfigurací nástroje Composer bude zajištěno použití pouze kompatibilních verzí knihoven. Nebude tak docházet k aktualizaci na verze obsahující zpětně nekompatibilní změny. Tento požadavek je obzvláště důležitý pro zajištění bezkonfliktního vývoje. Využití balíčkovacího systému přináší i tu výhodu, že vytvořená aplikace neobsahuje zdrojové kódy knihoven, které jsou balíčkovacím systémem spravovány. Odpadá tedy veškerá starost o manuální správu externích knihoven.

## 6.2 Automatický proces aktualizace

Automatický proces aktualizace tvoří sekvence operací, které mají za cíl reflektovat na cílový server změny provedené ve vyvíjené aplikaci.

V požadavcích je uvedeno, že proces aktualizace bude řízen pomocí služby Deploy. Služba Deploy umožňuje kromě nahrávání souborů také spouštět na cílovém serveru shellové skripty. Skripty lze spouštět před i po aktualizaci zdrojových souborů. Ve správci prováděných skriptů služby Deploy lze u skriptů nastavit, aby v případě chybového výstupu byla aktualizace ukončena. Tato možnost bude použita při aktualizaci struktury databáze. Pokud tato aktualizace selže, nemá smysl v aktualizaci pokračovat.

Pro zajištění pokrytí všech podpůrných procesů aktualizace je nutné vytvořit sadu shellových skriptů. Skripty budou navrženy tak, aby řešily vždy jen jeden stanovený úkol. Je zapotřebí vytvořit skripty pro:

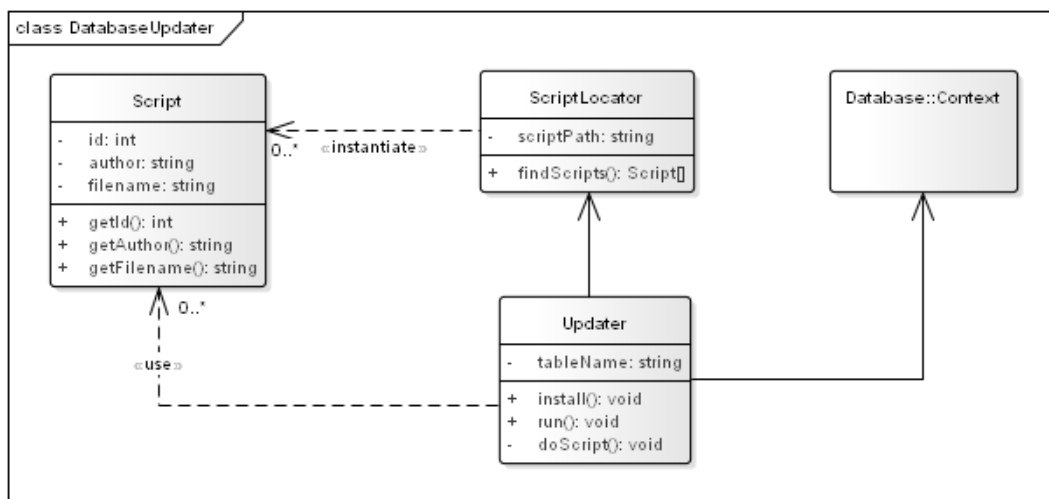
- zapnutí a vypnutí režimu maintenance
- vymazání obsahu paměti cache
- aktualizaci struktury databáze

### 6.2.1 Aktualizace struktury databáze

Nástroj pro aktualizaci struktury databáze bude sekvenčně zpracovávat frontu databázových skriptů. Fronta bude tvořena soubory uloženými ve speciálním adresáři. Pořadí ve frontě bude udávat celočíselný prefix v názvu souboru. Stejně číslo v názvu skriptu není možné použít vícekrát. Každý databázový skript bude vykonán pouze jednou. Soubor s databázovým skriptem obsahuje pouze čistý SQL kód. Proces aktualizace bude pouze jednocestný. Není tedy možné provádět downgrade na předchozí verzi. Pro aktualizací nástroj bude připraveno vhodné UI.

Diagram tříd uvedený na obrázku 6.1 zobrazuje návrh řešení. Třída *Updater* zajišťuje vykonání obsahu nalezených SQL skriptů. O nalezení skriptů z definovaného adresáře se stará třída *ScriptLocator*, která vrací kolekci objektů *Script*. Třída *Script* slouží jako DTO pro informace o souboru jako je identifikátor, název a cesta k souboru.

Model chování je uveden v sekvenčním diagramu na obrázku *updater:sequence*. Jádro tvoří iterace přes kolekci objektů *Script*. Tato kolekce byla získána pomocí metody *ScriptLocator::findScripts*. Při iteraci je každý prvek volána privátní metoda *Updater::doScript*, která zajišťuje vykonání nalezeného skriptu. Vykonání obsahu skriptu se provede pouze jednou. Při opětovném spuštění aktualizace bude vykonání obsahu skriptu přeskočeno.



Obr. 6.1 Diagram tříd procesu aktualizace struktury databáze

Pro implementaci navrženého řešení jsou připraveny jednotkové testy. Test *ScriptLocatorTest* je navržen pro validaci vyhledávání SQL skriptů. Je zde testován hlavní úspěšný scénář, scénář duplikátního identifikátoru souboru a také scénář nevalidního názvu SQL skriptu. Struktura testu hlavního scénáře má tuto podobu.

```

<?php
$expectedScripts = [
    new Script(1, 'johndoe_init', '001_johndoe_init.sql'),
    new Script(2, 'terminator_create', '2_terminator_create.sql'),
    new Script(20, 'johndoe_update', '020_johndoe_update.sql')
];

$scriptLocator = new ScriptLocator(__DIR__ . '/data/correct');
$scripts = $scriptLocator->findScripts();

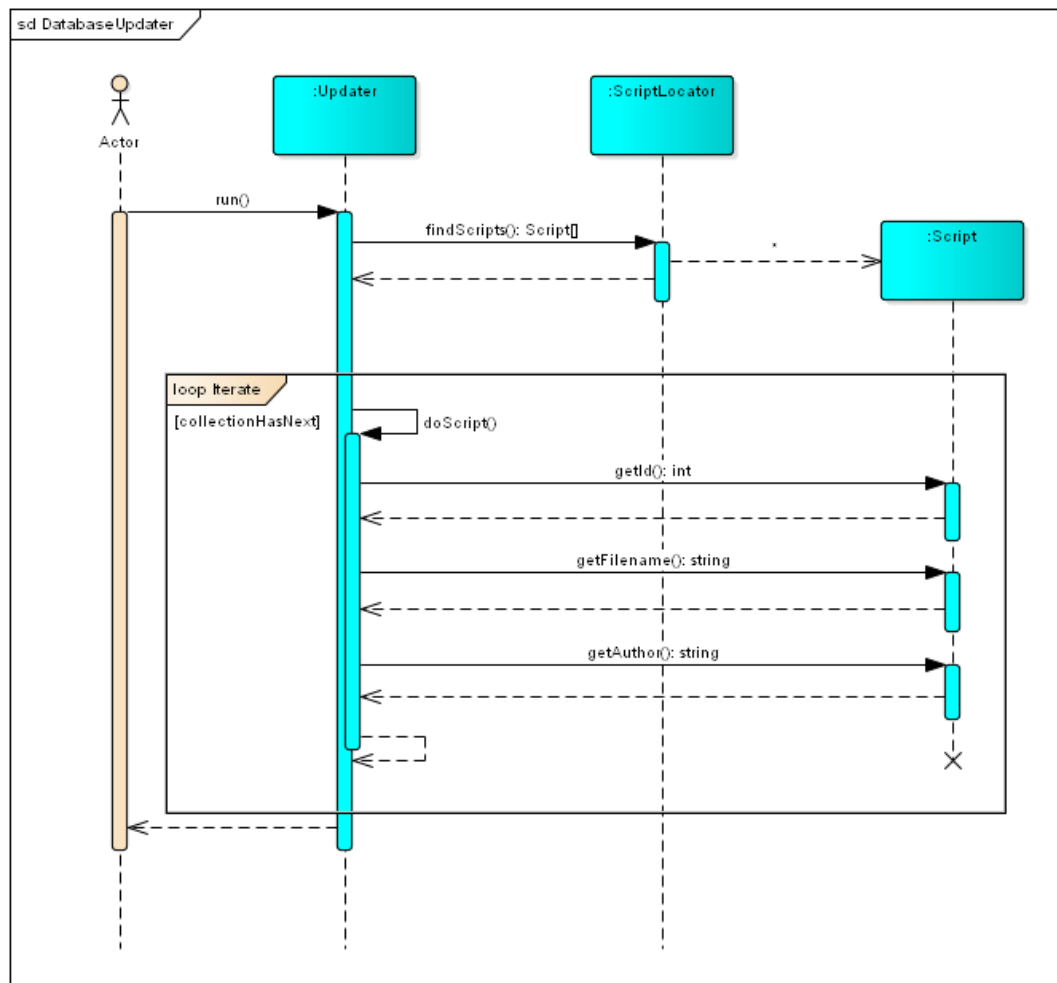
Tester\Assert::type('array', $scripts);
Tester\Assert::same(3, count($scripts));

for ($i = 0; $i < 3; $i++) {
    $expected = $expectedScripts[$i];
    $actual = $scripts[$i];

    Tester\Assert::same($expected->getId(), $actual->getId());
    Tester\Assert::same($expected->getAuthor(), $actual->getAuthor());
    Tester\Assert::same($expected->getFilename(), $actual->getFilename());
}

```

Test *UpdaterTest* je navržen pro validaci celého procesu nalezení SQL skriptů a zajištění jejich vykonání. Test je navržen tak, aby se testovaly očekávané vlastnosti



Obr. 6.2 Sekvenční diagram procesu aktualizace struktury databáze

samostatně. Nejprve se testuje zda se správně ukládá vnitřní pomocná struktura identifikující provedené skripty.

```

<?php
$expectedData = [
    ['id' => 1, 'file' => '001_johndoe_init.sql', 'author' => 'johndoe_init'],
    ['id' => 2, 'file' => '2_terminator_create.sql', 'author' =>
    ↵ 'terminator_create'],
    ['id' => 20, 'file' => '020_johndoe_update.sql', 'author' =>
    ↵ 'johndoe_update']
];

$scriptLocator = new ScriptLocator(__DIR__ . '/data/correct');
$updater = new Updater('versions', $scriptLocator, $context);
$updater->install();
$updater->run();

$table = $context->table('versions');
  
```

```
$actualData = [];  
  
foreach ($table->select('id, file, author')->fetchAll() as $row) {  
    $actualData[] = $row->toArray();  
}  
  
Assert::same(3, $table->count());  
Assert::same($expectedData, $actualData);
```

Dále se testuje úspěšné vykonání skriptů a také zda je obsah skriptu vykonán pouze jednou.

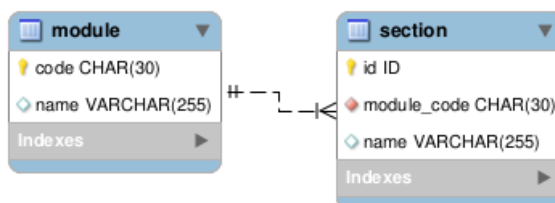
```
<?php  
$scriptLocator = new ScriptLocator(__DIR__ . '/data/correct');  
$updater = new Updater('versions', $scriptLocator, $this->context);  
$updater->install();  
$updater->run();  
  
Assert::noError(function () use ($updater) {  
    $updater->run();  
});  
  
$context->getStructure()->rebuild();  
$table = $context->table('test');  
$expectedData = [  
    ['id' => 1, 'name' => 'First query'],  
    ['id' => 2, 'name' => 'Second value'],  
    ['id' => 3, 'name' => 'Third value']  
];  
  
$actualData = [];  
  
foreach ($table->select('id, name')->fetchAll() as $row) {  
    $actualData[] = $row->toArray();  
}  
  
Assert::same(3, $table->count());  
Assert::same($expectedData, $actualData);
```

### 6.3 Tématické rozdělení obsahu

Tématické členění aplikace lze chápat jako rozdělení aplikace do modulů. Jednotlivé moduly zobrazují datový model stejným, či různým způsobem. Tématické členění také znamená, že datům je možné specifikovat pro jakou sekci jsou určeny. Tyto skutečnosti jsou zachyceny v doménovém modelu jako doménové prvky modul a sekce. Modul

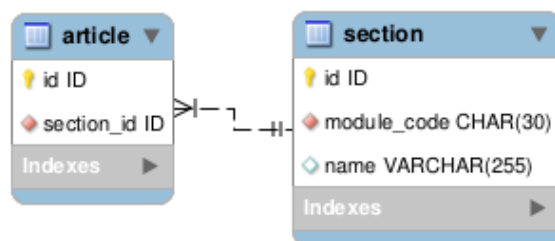
tvoří prezentační vrstvu, určuje jak se mají data zobrazovat. Sekce je definována jako identifikátor určující příslušnost dat. Framework Nette podporuje modulárnost aplikace a umožňuje tento koncept plně rozvinout. Sekce i modul tvoří doménové objekty a jsou modelovány jako databázové entity.

Na obrázku 6.3 je uveden navržený ER diagram sekcí. V diagramu jsou uvedeny dvě entity `section` a `module` s vazbou 1:N z pohledu entity `module`. Záznamy entity `module` budou obsahovat názvy modulů UI aplikace.



Obr. 6.3 ER diagram sekcí

Takto sestaveným modelem lze v systému realizovat více sekcí, přičemž různé sekce mohou používat stejný modul. Při tomto nastavení bude obsah obou sekcí stejně graficky interpretován ale v každé sekci budou zobrazeny jiné data. Takto lze například realizovat požadavek specifikující, že články jsou přiřazeny do jedné sekce. ER diagram příkladu je uveden na obrázku 6.4.

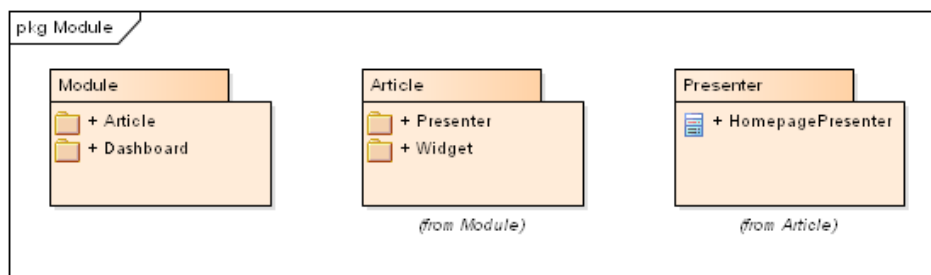


Obr. 6.4 ER diagram článku a sekce

Navržený model dle diagramu 6.3 realizuje nefunkční požadavek udržovatelnosti aplikace. Udržovatelnost je zaručena tím, že lze následně do systému kdykoliv doprogramovat nový modul.

Modul prezentační bude dále členěn na submoduly. Jednotlivé submoduly budou obsahovat presentery a další kódy potřebné pro realizaci dílčí části UI. Ukázka členění modulu je uvedena na obrázku 6.5. Příkladem submodulu je například submodul *Article*. Tento submodul obsahuje kódy pro realizaci výpisu článků, zobrazení obsahu jednoho článku a další případy užití.

V konfiguračním souboru aplikace je nutné nastavit pro každý vytvořený modul záznam do mapovací tabulky presenteru. Mapovací tabulka je využita třídou *Presen-*



Obr. 6.5 Diagram členění modulů

*terFactory*. *PresenterFactory* implementuje rozhraní *IPresenterFactory*. Toto rozhraní slouží k vytvoření instance presenteru na základě předaného parametru obsahujícího vnitřní adresu frameworku.

Mapování zajistí, že například adresa `:Module:Submodule:Dashboard` bude převedena na instanci třídy `App\Module\Submodule\Presenter\DashboardPresenter`. Nastavení mapování by pro tento příklad odpovídalo tomuto zápisu v konfiguračním souboru:

```

nette:
  application:
    mapping:
      Module: App\Module\*\Presenter\*Presenter
  
```

## 6.4 Strukturované URL

Požadavek strukturovaného URL plynule navazuje na požadavek rozdělení obsahu aplikace podle obsahu. Z technického hlediska jej lze rozdělit do dvou částí.

První část zajišťuje použití domény třetího řádu pro každou jednotlivou sekci.

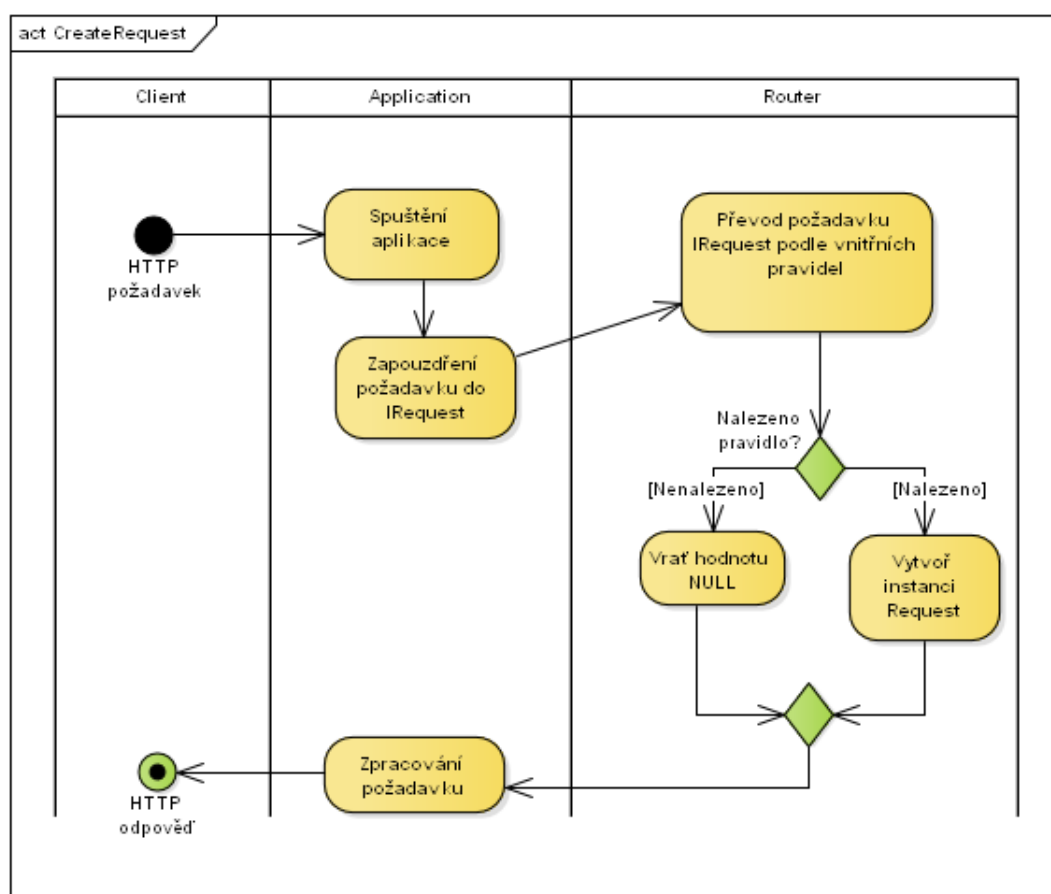
Druhá část zajišťuje strukturování URL podle obsahu aplikace. Výsledná struktura není v požadavku konkrétně zadána, je zde uvedeno pouze doporučení. Z obdobných projektů či nepsaných konvencí ji lze realizovat tak, že například pro výpisy článků bude užito vzoru:

- `example.cz\articles` pro výpis článků podle data publikace
- `example.cz\articles\author\john.doe` pro výpis článků pro autora John Doe
- `example.cz\articles\tag\ms2016` pro výpis článků podle značky ms2016

Oba požadavky strukturovaného URL jsou podporovány navrženou architekturou prezentační vrstvy. Presentery UI vrstvy využívají pro identifikaci sekce perzistentní identifikátor. Tento identifikátor musí být přítomen v každém URL odkazu. Přítomností identifikátoru v URL bude realizován požadavek různé domény třetího řádu pro

jednotlivé sekce. Všechny HTTP požadavky musí nejprve projít procesem překladač domény třetího řádu na identifikátor sekce. Zároveň při generování odkazů musí být identifikátor sekce přeložen na příslušnou doménu třetího řádu.

Zpracování HTTP požadavku pracuje dle diagramu uvedeném na obrázku 6.6. Z diagramu vyplývá, že každý HTTP požadavek zapouzdřený do implementace rozhraní *IRequest*, je převeden routerem (rozhraní *IRouter*, metoda *IRouter::match*) do instance třídy *Request*. Router je tedy zodpovědný za převod požadavku na vnitřní adresu frameworku určující konkrétní presenter. Router také umožňuje modifikovat parametry předávané presenteru. V neposlední řadě určuje podobu vytvářeného odkazu (metoda *IRouter::createUrl*).



Obr. 6.6 Aktivitní diagram zpracování HTTP požadavku

Vhodnou implementací routeru bude zajištěn obousměrný převod domény třetího řádu a identifikátoru sekce. Nastavenými pravidly routeru bude také realizován požadavek strukturovaného URL.

## 6.5 Databázová vrstva *nette/database*

Balíček *nette/database* zajišťuje přístup do relační databáze (DBMS) podle návrhového vzoru Table/Row Data Gateway. Neobsahuje ORM vrstvu, která by zajišťovala mapování mezi doménovými objekty a entitami relační databáze. Obsahuje však vrstvu DBAL, která umožňuje využití více různých DBMS. Model databáze proto bude popsán pomocí ER diagramu, který bude zachycovat entity s atributy včetně příslušných relací. Business logika bude implementována do doménového modelu pomocí doménových služeb.

Balíček *nette/database* (konkrétně vrstva Table), je určen pro zjednodušený výběr dat z relační databáze. Zjednodušeným výběrem je myšleno, že při běžném použití není definováno mapování mezi entitami v databázi. Vztahy mezi entitami si vrstva Table zjišťuje sama podle struktury realizované cizími klíči. Zjednodušeno je i zadávání příkazů selekce a projekce.

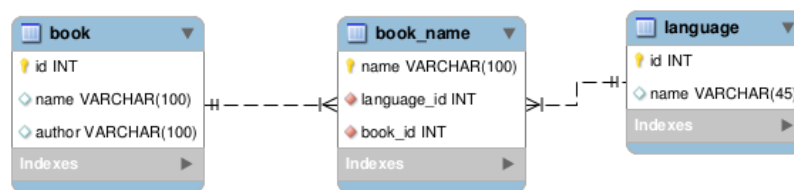
Využití vrstvy Table je však omezeno na konkrétní styl práce. Především se jedná o sestavování jednoduchých dotazů a následné postupné iterace přes získaný výsledek. Tento způsob práce však není vhodný pro rozsáhle aplikace, kde je nutno počítat s komplexními dotazy. Komplexní dotazy jsou nutné pro zajištění vysoce optimalizovaného získání konkrétních dat vyžadovaných prezentační vrstvou. Řešení lze realizovat pomocí využití pohledů (view) databáze, nebo využít vrstvu Table způsobem, ke kterému nebyla navržena. Použití alternativního způsobu práce s vrstvou Table vyžaduje nalézt řešení pro problémy, které tento způsob využití obnáší.

Prvním problémem využití alternativního způsobu práce je to, že při výběru dat z více než jedné tabulky dochází k využití klauzule LEFT JOIN výsledného SQL příkazu, bez možnosti specifikovat parametry klauzule ON. Realizace komplexních dotazů však vyžaduje schopnost tyto parametry specifikovat. Parametry selekce je možné zadat do klauzule WHERE, nicméně v tomto případě je selekce příliš striktní. Pokud data v připojené tabulce nebudou odpovídat kritériu selekce, bude výsledkem celého dotazu prázdný výsledek. Nelze tedy využít schopnosti, které relace LEFT JOIN nabízí.

Na příkladu je ilustrována modelová situace, kde existuje tabulka knih. Každá kniha obsahuje název a autora. Kniha může obsahovat jeden překlad názvu pro každý jazyk. V testu uvažujeme výběr všech záznamů z tabulky knih včetně překladu do konkrétního jazyka. Alternativní název knihy je však nepovinný - může být NULL. Struktura databáze je uvedena na obrázku 6.7.

Tabulky jsou naplněny pomocí tohoto SQL skriptu:

```
INSERT INTO 'book' ('id', 'name', 'author') VALUES
(1, 'Enterprise patterns', 'John Doe'),
(2, 'Usefull examples', 'Johny English'),
```



Obr. 6.7 ER diagram příkladu databáze knih

```
(3, 'Domain driven design', 'Evans');
```

```
INSERT INTO 'language' ('id', 'name') VALUES
```

```
(1, 'Czech'),
```

```
(2, 'English');
```

```
INSERT INTO 'book_name' ('name', 'language_id', 'book_id')
```

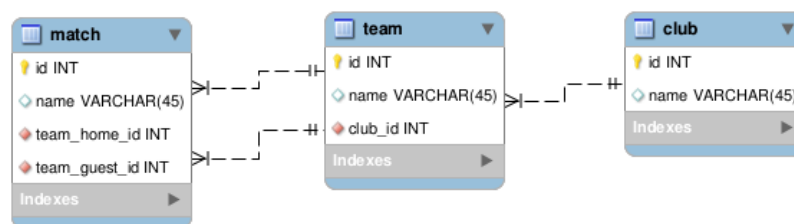
```
VALUES ('Uzitecne priklady', 1, 2),
```

```
('Usefull examples', 2, 2);
```

Test ověřující očekávaný výsledek má tuto podobu:

```
<?php
$select = $this->context->table('book')
->select('book.name, :book_name.name AS translated_name')
->where(':book_name.language.id OR :book_name.name IS NULL', 1);
Assert::same(3, $select->count());
```

Druhým problémem při využití alternativního způsobu práce naráží na fakt, že pro připojené tabulky nelze specifikovat databázový alias. Databázový alias je nutné využít v případě, že v SQL dotazu je jedna entita připojena dvakrát v pro dvě různé podmínky. Na příkladu je ilustrována modelová situace, kde existuje záznam zápasu. Zápas obsahuje dvě vazby do tabulky týmů. První vazba představuje domácí tým, druhá tým hostů. Každý tým je součástí jednoho klubu. V testu je uvažována situace, kde jsou požadovány informace o zápase a zároveň jména domácího a hostujícího klubu. Struktura databáze je uvedena na obrázku 6.8.



Obr. 6.8 ER diagram příkladu databáze zápasu

Tabulky jsou naplněny pomocí tohoto SQL skriptu:

```
INSERT INTO 'club' ('id', 'name') VALUES
(1, 'Club 1'),
(2, 'Club 2'),
(3, 'Club 3');

INSERT INTO 'team' ('id', 'name', 'club_id') VALUES
(1, 'Team 1', 1),
(2, 'Team 2', 2),
(3, 'Team 3', 3);

INSERT INTO 'match' ('id', 'name', 'team_home_id', 'team_guest_id') VALUES
(1, 'Match 1', 1, 2),
(2, 'Match 2', 3, 1);
```

Test ověřující očekávaný výsledek má tuto podobu:

```
<?php
$row = $this->context->table('match')
->select('match.name, .team_home.club.name AS club_home_name,
↪ .team_guest.club.name AS club_guest_name')
->where('match.id', 1)
->fetch();
Assert::same('Club 1', $row['club_home_name']);
Assert::same('Club 2', $row['club_guest_name']);
```

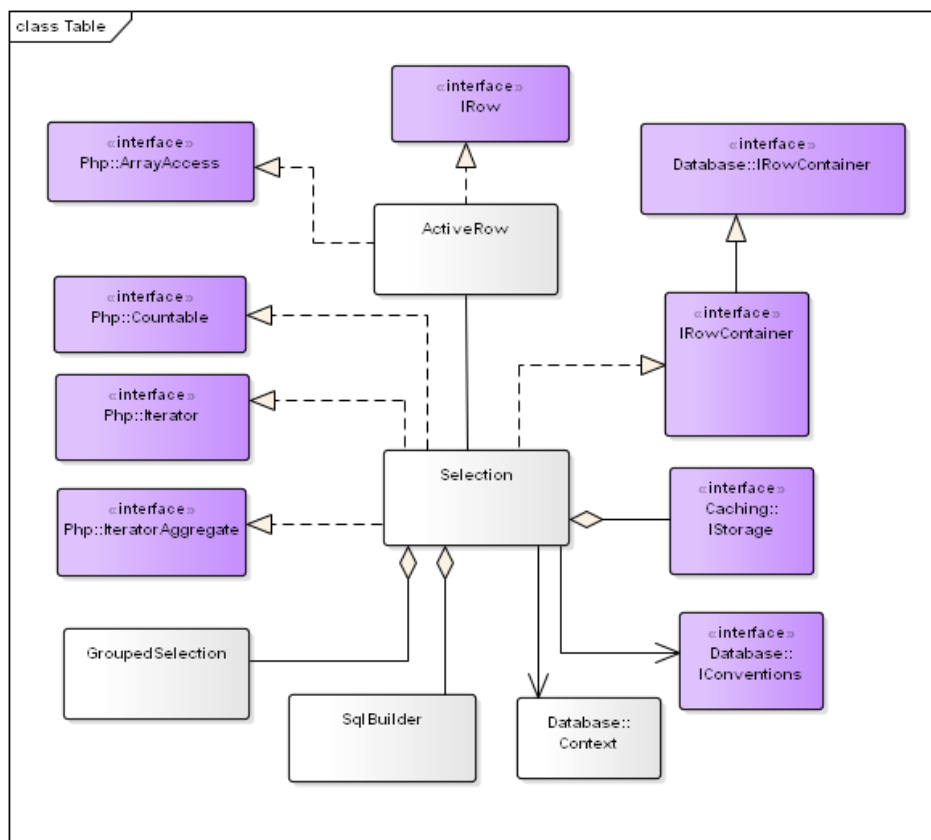
Alternativní způsob využití vrstvy Table vyžaduje doplnit metody, které by umožnily realizovat situace popsané v příkladech.

Struktura vrstvy Table je zachycena diagramem tříd uvedeným na obrázku 6.9. Z diagramu je zřejmé, že systém je navržen jako monolyt a nelze do něj jednoduše přidat nové funkce a chování. Bude tak nutné celý balíček reimplementovat formou nového balíčku. Nový balíček přitom nahradí balíček původní. Pro specifikaci nových funkcí a popis jejich chování, jsou vytvořeny jednotkové testy.

Ukázka z testů připravených pro funkci left:

```
<?php
test(function() use ($context) {
    $sql = $context->table('book')->left('translator.name',
↪ 'Geek')->select('book.*')->getSql();
    Assert::same(reformat('SELECT [book].* FROM [book] LEFT JOIN [author] [translator]
↪ ON [book].[translator_id] = [translator].[id] AND ([translator].[name] = ?)'),
↪ $sql);
});
```

Ukázka z testů připravených pro funkci alias:



Obr. 6.9 Diagram tříd struktury vrstvy Table balíčku nette/database

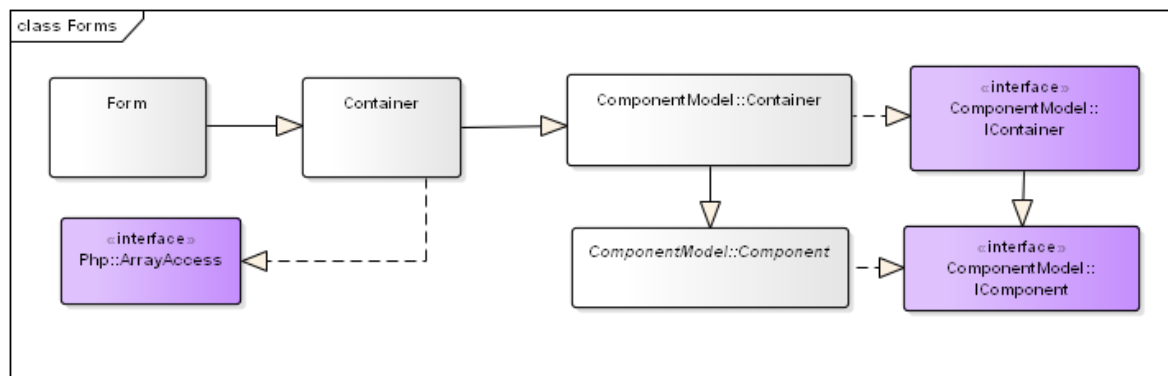
```

<?php
test(function() use ($context) {
    $sql = $context
        ->table('book')
        ->alias('translator', 'trans')
        ->left('trans.name', 'Geek')
        ->select('book.*')
        ->getSql();
    Assert::same(reformat(
        'SELECT [book].* FROM [book] '
        . 'LEFT JOIN [author] [trans] ON [book].[translator_id] = [trans].[id] AND
        ↪ ([trans].[name] = ?)'), $sql);
});
  
```

## 6.6 Dynamické formuláře

Pro práci s formuláři je ve frameworku Nette určen balíček *nette/forms*. Formuláře využívají komponentní model frameworku Nette. Formulář i jeho prvky implementují rozhraní *IComponent*. Balíček obsahuje třídu *Container* rozšiřující třídu *ComponentModel\Container*. Tato třída tvoří těžiště implementace formuláře a všech formulářo-

vých prvků. Diagram tříd implementace formulářů je uveden na obrázku 6.10.



Obr. 6.10 Diagram tříd komponenty formuláře z balíčku *nette/forms*

Balíček *nette/forms* obsahuje metody podporující tvorbu pouze statických formulářů ve smyslu pevně stanoveného počtu prvků formuláře. Dynamická podoba formuláře má své opodstatnění. V případě kdy například existuje formulář zápasu, ve kterém jsou také uvedeny data ke hráčům, kteří v zápase nastoupili. Počet hráčů v zápase není omezen. Standartní realizace by mohla vypadat následovně.

```

<?php
$form = new Form();
$roster = $form->addContainer('roster');
foreach ([1, 2, 3] as $id) {
    $roster->addText("jersey_{$id}", 'Číslo dresu');
    $roster->addText("goal_{$id}", 'Počet gólů');
    $roster->addText("yellow_{$id}", 'Počet žlutých karet');
}

```

Jasnou nevýhodou tohoto řešení je vložení identifikátoru do názvu formulářového prvku ve formě postfixu v názvu formulářového prvku. Identifikátor záznamu lze od názvu formulářového prvku oddělit využitím dalšího kontejneru. Výsledek je uveden na příkladu.

```

<?php
$form = new Form();
$roster = $form->addContainer('roster');
foreach ([1, 2, 3] as $id) {
    $playerContainer = $roster->addContainer($id);
    $playerContainer->addText("jersey", 'Číslo dresu');
    $playerContainer->addText("goal", 'Počet gólů');
    $playerContainer->addText("yellow", 'Počet žlutých karet');
}

```

Využití samostatného kontejneru pro uložení identifikátoru je vhodná cesta k realizaci dynamických formulářů. Implementace UI předpokládá využití dynamických formulářů na mnoha místech aplikace, proto je vhodné vyčlenit obslužný kód pro vytvoření kontejneru do samostatné metody. Použití dynamického formuláře bude probíhat podle uvedeného příkladu. Prvním parametrem metody *Container::addDynamicContainer* je název kontejneru, druhým je odkaz na funkci plnící obsah kontejneru.

```
<?php
$form = new Form;
$form->addDynamicContainer('rows', function () {
    $container = new Nette\Forms\Container;
    $container->addText("jersey", 'Číslo dresu');
    $container->addText("goal", 'Počet gólů');
    $container->addText("yellow", 'Počet žlutých karet');
    return $container;
});
$form->setItems([1, 2, 3]);
```

Dynamický kontejner vytvořený podle výše uvedeného popisu není schopen realizovat požadavek zadání uvádějící, že obsah kontejneru je možné měnit v závislosti na změně hodnoty některého z prvků formuláře. Typickým příkladem je formulář pro zápis statistik hráče. U hráče je k dispozici výběr herní pozice. Změna hodnoty tohoto prvku má za důsledek změnu složení formuláře pro zápis statistiky. Pro brankáře jsou totiž definovány jiné údaje než pro ostatní hráče. Navržená realizace bude odpovídat obdobnému způsobu použití jako v minulém příkladě. Funkce předávaná kontejneru bude uvnitř rozhodovat na základě podmínky o obsahu kontejneru. Následující příklad ilustruje očekávaný způsob použití.

```
<?php
$form = new Form;
$form->addSelect('options', 'Výběr', ['article' => 'Článek', 'news', 'Novinka']);
$form->addLazyContainer('items', function (LazyContainer $self) {
    $values = $self->getFormValues();
    $this->setupContainerByType($self, $values['options']);
});
```

Třetím typem dynamického prvku je dle zadání formulářový prvek podporující UI knihovnu Select2. Knihovna Select2 slouží k vytváření inteligentních formulářových prvků typu selectbox. Knihovna Select2 implementuje funkcionalitu požadovanou zadáním tak, že zneviditelní vybraný formulářový prvek typu `input[type=text]` a na jeho místo vloží inteligentní selectbox. Prvky selectboxu jsou vytvořeny rozdělením obsahu zneviditelněného pole formuláře pomocí nastaveného oddělovače. Po změně obsahu

inteligentního selectboxu je zneviditelněnému prvku nastavena hodnota formou textového spojení položek selectboxu nastaveným oddělovačem. Očekávaný způsob práce s navrhovaným formulářovým prvkem je uveden na následujícím příkladu.

```
<?php
$form = new Form;
$form->addMultiselect2('tags', 'Tagy');
```

Pro implementaci metod dle výše popsaného chování jsou navrženy testy, které ověří výsledek implementace. Ukázka z navržených testů je uvedena níže.

```
<?php
test(function(){
    $expected = [
        5 => ['name' => 'John', 'surname' => 'Doe'],
        10 => ['name' => 'Johny', 'surname' => 'English']
    ];

    $multi = new MultiContainer(function($name, $items){
        $container = new \Nette\Forms\Container;
        $container->addText('name');
        $container->addText('surname');
        $container->setDefaults($items);
        return $container;
    });

    $multi->setItems($expected);
    Assert::same($expected, $multi->getValues(true));
});

test(function(){
    $form = \Mockery::mock('Nette\Forms\Form');
    $form->shouldIgnoreMissing();
    $form->shouldReceive('isSubmitted')
        ->andReturn(false);
    $form->shouldReceive('isAnchored')
        ->andReturn(true);
    $form->shouldReceive('getValues')
        ->andReturn(['type' => 'article']);

    $multi = new LazyContainer(function(LazyContainer $self){
        $values = $self->getFormValues();
        if ($values['type'] == 'article') {
            $self->addText('title');
        } else {
```

```
        $self->addText('name');
    }
});

$multi->setParent($form);
$multi->setDefaults(['title' => 'Article name']);
Assert::same(['title' => 'Article name'], $multi->getValues(true));
});

test(function(){
    $input = new Multiselect2Input('a');
    $input->setValue('a,b,c,d');
    Assert::same(['a', 'b', 'c', 'd'], $input->getValue());
});
```

## 6.7 Flexibilní modelová vrstva

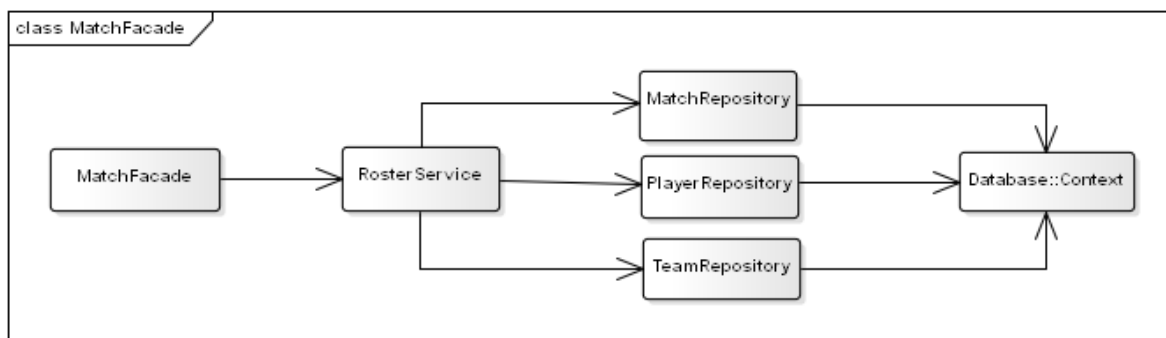
Flexibilní modelová vrstva je z pohledu zadání taková vrstva, která zajistí dobrou čitelnost, přehlednost, konzistenci a udržitelnost. Návrh uvažuje vybrané vlastnosti přístupu Command Query Separation. Velmi okrajově je uvažován také Domain Driven Design. Značnou roli v návrhu hraje zvolený způsob perzistence dat pomocí *nette/database*. V návrhu hraje svou roli i plnění nefunkčních požadavků v podobě co nejnižšího zatížení databáze, zajištění konzistence dat a nízké paměťové režie. Návrh využívá znalosti a postupy, které byly získány v průběhu realizace obdobných systémů.

Navržená modelová vrstva nebude využívat doménové třídy, které jsou obvykle základem modelové vrstvy. Modelové třídy nejsou užity vzhledem k použitému způsobu přístupu do relační databáze realizovaného vrstvou Database frameworku Nette. Doménová logika bude zakomponována do doménových služeb.

Vnější rozhraní modelu budou tvořit fasády. Prostřednictvím fasád budou realizovány uživatelské scénáře. Fasády budou tyto scénáře realizovat prostřednictvím doménových služeb. Skrze doménové služby budou také realizovány všechny business procesy. Doménové služby budou využívat služeb repositářů. Repozitář bude tvořit sada metod zajišťujících perzistenci a výběr dat z jedné databázové tabulky.

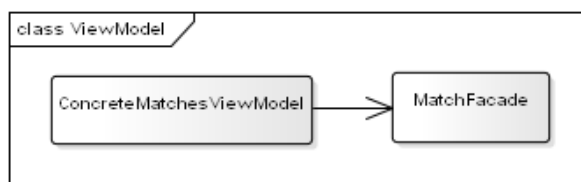
Fasády prostřednictvím svých metod budou primárně učeny pro modifikaci domény. Pro účely čtení dat z domény budou vytvořeny ViewModely. ViewModel bude třída poskytující data ve struktuře, kterou vyžaduje konkrétní pohled View vrstvy. Tyto ViewModely budou pro výběr dat využívat optimalizovaný přístup spočívající v sestavení jednoho konkrétního dotazu do databáze (prostřednictvím vrstvy Table).

Na obrázku 6.11 je uveden příklad diagramu tříd znázorňující strukturu modelové vrstvy pro realizaci požadavku přidání soupisky do zápasu.



Obr. 6.11 Diagram tříd vzorové struktury modelové vrstvy

Obrázek 6.12 obsahuje diagram tříd znázorňující příklad ViewModelu.



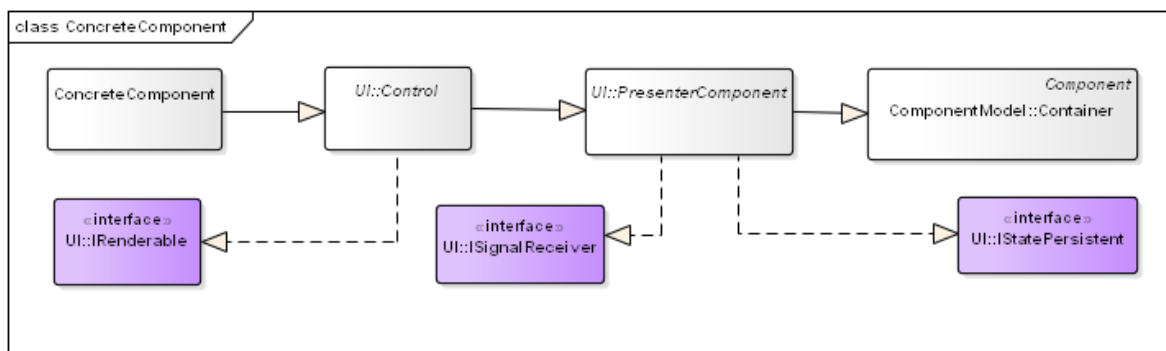
Obr. 6.12 Diagram tříd vzorového view modelu

## 6.8 Znovupoužitelné komponenty uživatelského rozhraní

Komponenty prezentační vrstvy jsou součástí uživatelského rozhraní a slouží k převodu informací do podoby určené pro grafické rozhraní klienta, které následně informace interpretuje do grafické podoby. Pro webové aplikace je výstupním formátem značkovací jazyk HTML.

UI komponenty využívané aplikací vyžadují specifikovat překladač statických textů (rozhraní *ITranslator*) pro zajištění aktuální verze textových překladů a pro zajištění správného skloňování výrazu množství. Aplikace využívá množství helperů systému Latte. Pro registraci helperů do šablony je využita komponenta *HelperLoader*, kterou je nutné injektovat do každé vytvořené instance UI komponenty. Komponenty vložené do šablonovacího systému Latte makrem *control* vyžadují, aby každá komponenta obsahovala metodu *render*. Metoda *render* zajistí renderování šablony komponenty.

Komponenty UI lze ve frameworku Nette realizovat pomocí třídy *Control* z balíčku *nette/application*. Komponenty budou používat abstraktního popisu zdroje zobrazovaných dat. Abstraktní zdroj dat bude definován pomocí rozhraní (interface). Konkrétní datový zdroj bude ve většině případů ViewModel z modelové vrstvy. Příklad komponenty ilustruje diagram tříd na obrázku 6.13.

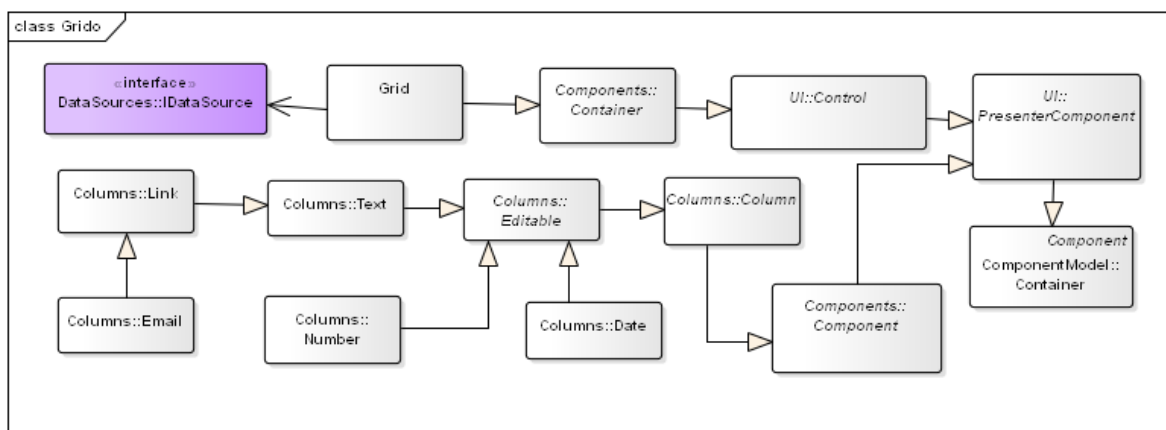


Obr. 6.13 Diagram tříd vzorové komponenty

## 6.9 Datové gridy s možností inline editace

Pro realizaci datových gridů bude využita komponenta Grid balíčku *o5/grido*. Grid splňuje všechny požadavky stanovené v zadání mimo požadavku přímé inline editace.

Vnitřní struktura komponenty Grid je zachycena diagramem tříd na obrázku 6.14. Z diagramu lze vyčíst, že buňky datového gridu jsou realizovány jako objekty specializující třídu *PresenterComponent*. *PresenterComponent* je navržen jako prezistentní objekt umístěný v presenteru a dovoluje interakci s uživatelem. Z této skutečnosti vyplývá, že každá buňka datového gridu umožňuje přímou interakci s uživatelem. Implementace inline editace bude spočívat ve tvorbě interakčního UI prvku a zpracování události, která bude vyvolána změnou obsahu editovaného prvku.



Obr. 6.14 Diagram tříd komponenty Grid

Jako interakční prvky uživatelského rozhraní budou použity standartní formulářové prvky typu text, checkbox a select. Rozhraní pro definici sloupců s podporou inline editace bude odpovídat následující podobě:

```
<?php
use Grido\Grid;
$callback = function ($id, $value) {};
```

```
$grid = new Grid;
$grid->addColumnInput('name', 'Název', $callback);
$grid->addColumnSelect('code', 'Name', $callback)
    ->setItems($arrayOfValues);
$grid->registerColumnSwitch('active', 'Aktivní', $callback);
```

Proměnná *\$callback* je odkaz na funkci, která realizuje změnu hodnot v modelu. Callback obdrží dva parametry. První parametr bude hodnota identifikátoru záznamu. Druhým parametrem bude hodnota zadaná uživatelem prostřednictvím interakčního prvku. Pole *\$arrayOfAllowedValues* obsahuje seznam hodnot nabízených uživateli v interakčním prvku typu selectbox.

## 6.10 Systém řízení uživatelských oprávnění

Pro splnění požadavku realizujícího zabezpečení aplikace, je nutné nejprve identifikovat a sestavit modely uživatele, autorizace a autentifikace. Framework Nette obsahuje model uživatele včetně jeho implementace v balíčku *nette/security*. Model uživatele reprezentovaný třídou *User* je využíván všemi ostatními částmi frameworku. Odkazuje se na ni například standartní implementace presenteru a továrny pro tvorbu šablon. Z těchto důvodů bude nejvhodnější tento model uživatele převzít a jeho subkomponenty implementovat podle specifikace frameworku.

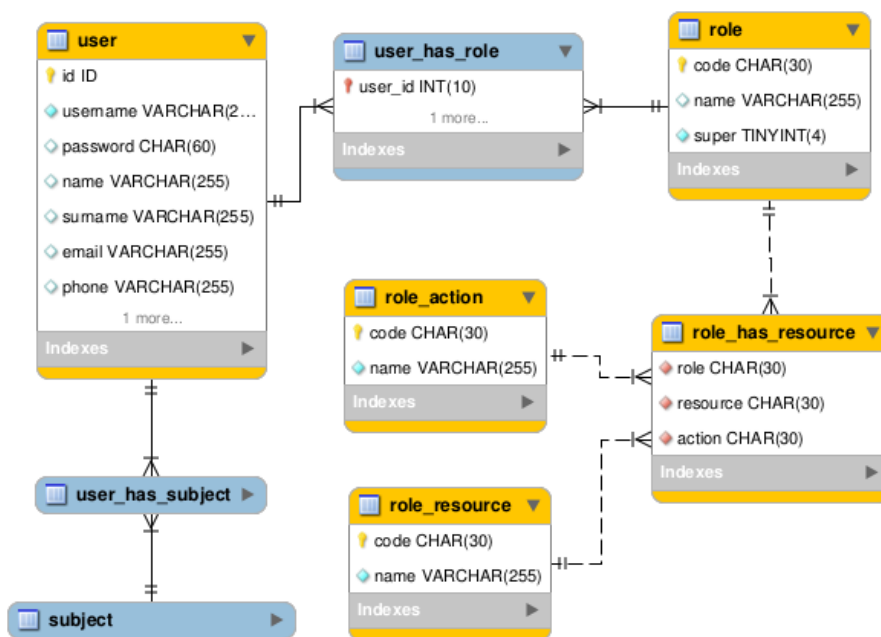
Diagram tříd popisující model uživatele frameworku Nette je uveden na obrázku 2.2 v teoretické části této práce. Z diagramu vyplývá skutečnost, že pro realizaci procesu autentifikace a autorizace, je nutné implementovat rozhraní *IA Authenticator* a *IA Authorizator*. Implementace těchto rozhraní proběhne podle stanovených uživatelských scénářů.

Prvním krokem návrhu je vytvoření ER modelu tabulek databáze. ER model je využit z důvodu, že nelze sestavit diagram tříd, neboť není použito mapování relační databáze na doménové třídy. ER diagram na obrázku 6.15 obsahuje návrh entit pro realizaci zabezpečení a vazeb mezi těmito entitami.

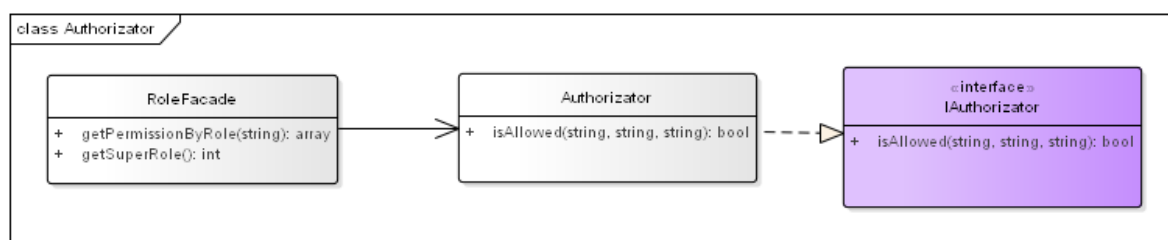
Z ER diagramu je patrná M:N vazba mezi uživatelem a uživatelskou rolí. Uživatelská role obsahuje M:N vazbu mezi zdrojem a akcí. Parametry role, zdroj a akce jsou vstupními parametry rozhraní metody *IAuthorizator::authorize*. Role obsahuje atribut *super*. Tento atribut symbolizuje superuživatelskou roli, pro kterou bude libovolný požadavek vždy autorizován.

Návrh implementace rozhraní *IAuthorizator* je uveden v diagramu tříd na obrázku 6.16. Třída *RoleFacade* poskytuje skrze své metody data potřebná pro zjištění nastavení zdrojů a akcí uživatelské role a taky poskytuje seznam všech superrolí.

Rozhraní *IA Authenticator* slouží k získání uživatelské identity definované rozhraním

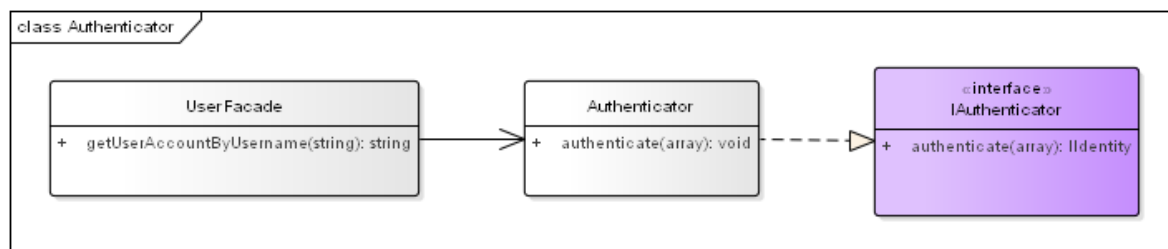


Obr. 6.15 ER diagram modelu zabezpečení



Obr. 6.16 Diagram tříd implementace rozhraní IAuthorizator

*Identity*. Návrh implementace rozhraní *IAuthenticator* je uvedeno v diagramu tříd na obrázku 6.17. Návrh definuje třídu *UserFacade* poskytující uživatelské data uložená v databázi.



Obr. 6.17 Diagram tříd implementace rozhraní IAuthenticator

Implementací výše popsaných rozhraní bude zajištěna schopnost autentifikovat uživatele a autorizovat jeho požadavky. Pro kompletní řízení uživatelských oprávnění je nutné zajistit, že každý uživatelův požadavek bude autorizován. Model uživatelský rolí a jejich oprávnění v kombinaci s návrhem UI dovoluje konstatování, že vnitřní adresa

frameworku Nette jednoznačně identifikuje prováděnou operaci, kterou lze podrobit procesu autorizace. Autorizaci požadavku lze vzhledem k využití výchozí implementace presenteru (rozhraní *IPresenter*). Ideálním místem v životním cyklu presenteru bude metoda *startup*. Vnitřní adresa je definována názvem presenteru, akcí a signálem. K realizaci řešení převodu vnitřní adresy na parametry resource a privilege potřebné k autorizaci požadavku je nutné vytvořit vhodný systém.

### 6.10.1 Systém pro převod vnitřní adresy na parametry ACL

Systém pro převod adresy bude na vstupu pracovat se třemi parametry identifikující konkrétní místo v aplikaci. Výsledkem převodu budou dvě hodnoty identifikující zdroj a akci. Tyto hodnoty budou použity při autorizaci požadavku.

Navržený systém bude překládat vstup pomocí konfiguračních pravidel. Tyto pravidla budou zapsána v konfiguračním souboru. Konfigurace bude rozčleněna na tři sekce. V konfiguraci lze využít zástupný znak \*. Tento znak symbolizuje jakýkoliv řetězec.

Převod adresy bude probíhat ve třech krocích. Výsledný systém bude složen ze samostatných komponent. V prvním kroku se generalizuje signál. Generalizace signálu spočívá v převodu signálu na identifikátor ve formě textového řetězce. Tento identifikátor pak bude použit v dalších dvou konfiguračních tabulkách pro identifikaci signálu. Konfigurace generalizace bude zapsána formou klíč - hodnota. Klíč bude složen ze dvou částí oddělených čárkou. První část klíče udává název příjemce signálu, druhá pak název signálu předávaného příjemci. Hodnota konfiguračního pravidla odpovídá názvu generalizovaného signálu.

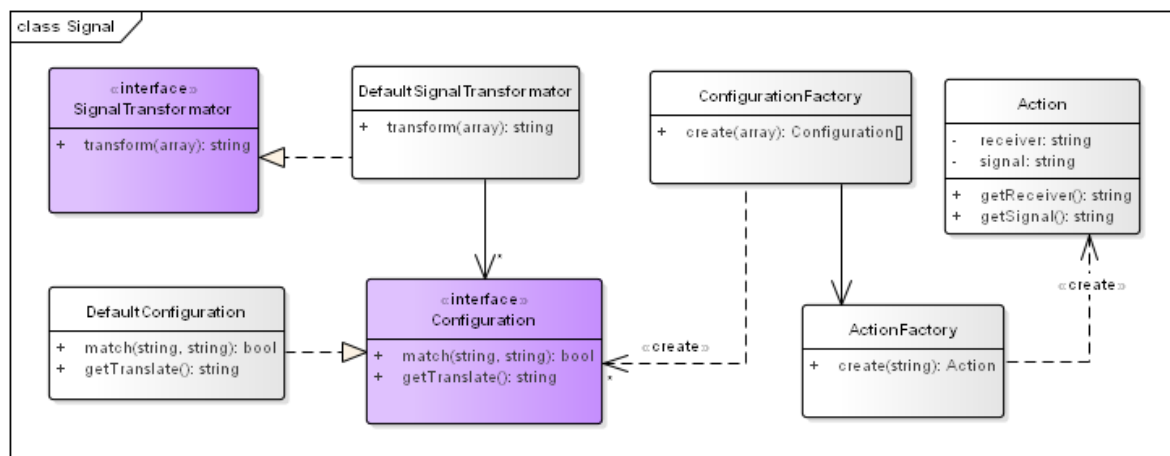
Diagram tříd komponenty pro převod signálu je uveden na obrázku 6.18. Návrh počítá s rozdělením na dvě části. První část zajistí převod konfiguračních pravidel do vhodné podoby, druhá část zajišťuje převod vstupních parametrů za pomoci konfiguračních pravidel na odpovídající výstup.

Konfigurační pravidla pro test převodu jsou zapsány v tomto tvaru.

```
<?php
$configuration = [
    '*grid-actions-delete,click' => 'delete',
    '*grid-form,submit' => 'view',
    "*,*" => "edit"
];
```

Testovací pravidla převodu jsou.

```
<?php
Assert::same('delete', $transformator->transform(['grid-actions-delete', 'click']));
```



Obr. 6.18 Diagram tříd komponenty převodu signálu

```
Assert::same('edit', $transformator->transform(['some-grid-actions-delete',
    ↪ 'submit']));
Assert::same('view', $transformator->transform(['some-grid-form', 'submit']));
Assert::same('edit', $transformator->transform(['some-receiver', 'some-signal']));
```

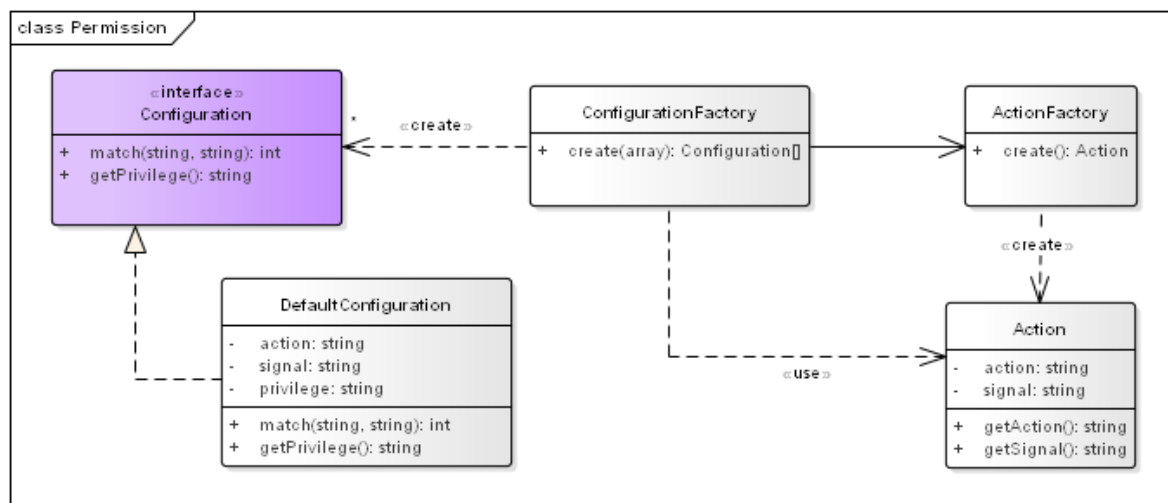
Druhý krok převodu adresy spočívá v nalezení konkrétního konfiguračního pravidla odpovídajícího vstupu. V tomto kroku se hledá odpovídající pravidlo vztažené na všechny tři vstupní parametry (presenter, akce a signál). Konfigurační pravidlo je zapsáno formou klíč - hodnota. Klíč je složen ze dvou částí oddělených čárkou. První část klíče tvoří celá vnitřní adresa frameworku Nette. Jedná se tedy o název presenteru a akce. Hodnota akce nemusí být specifikována - pak odpovídá libovolné akci. Druhá část klíče odpovídá hodnotě generalizovaného signálu. Tato část je nepovinná - pak vyhovuje libovolný signál. Hodnota konfiguračního pravidla je složena ze dvou částí oddělených čárkou. První část odpovídá názvu zdroje. Druhá část odpovídá názvu výstupní akce a její hodnota je v konfiguračním pravidlu nepovinná.

Pokud je výsledkem převodu pomocí této komponenty nekompletní informace (druhý parametr hodnoty konfiguračního pravidla nebyl zadán), dokončení převodu vstupní adresy provede třetí komponenta.

Diagram tříd komponenty převodu presenteru je uveden na obrázku 6.19.

Konfigurační pravidla pro test převodu jsou zapsány v tomto tvaru.

```
<?php
$arrayOfConfiguration = $factory->create([
    "Admin:Players:Roster:default,submit" => "Roster:edit",
    ":",submit" => ":edit",
    "Admin:Player:Bio:" => "Dial:",
    "Admin:Players:Rosters:,submit" => "Players:edit",
    "Admin:Players:*:" => "Dials:",
```



Obr. 6.19 Diagram tříd komponenty převodu presenteru

```

    "Admin:Players:*:default,submit" => "Dials:"
  });

```

Testovací pravidla převodu jsou.

```
<?php
```

```

$configuration = $arrayOfConfiguration[0];
Assert::true($configuration->match('Admin:Players:Roster', 'default', 'submit'));
Assert::false($configuration->match('Admin:Players:Roster', 'default', null));
Assert::false($configuration->match('Admin:Players:Roster', 'edit', 'submit'));

$configuration = $arrayOfConfiguration[1];
Assert::true($configuration->match('Admin:Articles:Homepage', 'default', 'submit'));
Assert::false($configuration->match('Admin:Articles:Homepage', 'default', null));

$configuration = $arrayOfConfiguration[2];
Assert::true($configuration->match('Admin:Player:Bio', 'default', 'submit'));
Assert::false($configuration->match('Admin:Articles:Homepage', 'default', null));

$configuration = $arrayOfConfiguration[3];
Assert::true($configuration->match('Admin:Players:Rosters', 'default', 'submit'));
Assert::false($configuration->match('Admin:Players:Rosters', 'default', null));

$configuration = $arrayOfConfiguration[4];
Assert::true($configuration->match('Admin:Players:Rosters', 'default', 'submit'));
Assert::true($configuration->match('Admin:Players:Bio', 'default', null));
Assert::false($configuration->match('Admin:Articles:Players', 'default', null));

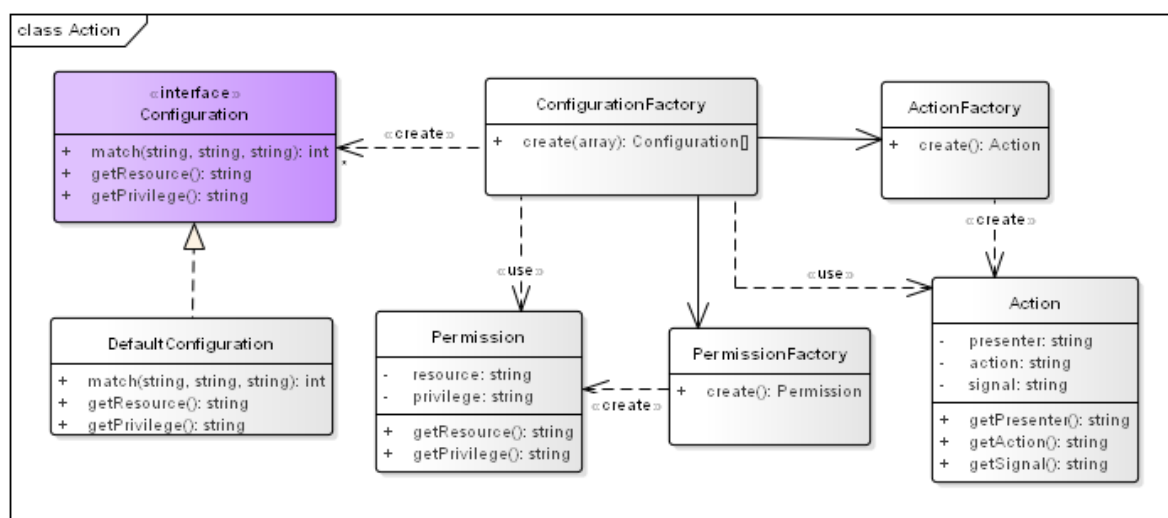
$configuration = $arrayOfConfiguration[5];
Assert::true($configuration->match('Admin:Players:Rosters', 'default', 'submit'));
Assert::false($configuration->match('Admin:Players:Rosters', 'edit', 'submit'));

```

```
Assert::false($configuration->match('Admin:Players:Bio', 'default', null));
```

Třetí krok převodu probíhá pouze, pokud není v druhém kroku nalezena kompletní informace. Smyslem třetí části je standardizovat převod parametru vstupní akce a signálu na konkrétní hodnotu parametru výstupní akce. Aplikační část celého systému totiž předpokládá, že například akce default odpovídá výpisu informace, add odpovídá vytvoření nového záznamu a edit editaci záznamu. Z těchto důvodů lze konfigurační pravidlo zjednodušit. Konfigurační pravidla komponenty pro převod akce je zapsána formou klíč - hodnota. Klíč je složen ze dvou částí oddělených čárkou. První část klíče odpovídá hodnotě vstupního parametru akce. Druhá část klíče je tvořena hodnotou normalizovaného signálu. Druhá část nemusí být uvedena - pak je uvažováno, že na na vstupu nebyl signál (symbolizováno hodnotou signálu NULL), přičemž oddělovač musí být v konfiguračním pravidlu přítomen.

Diagram tříd komponenty převodu akce je uveden na obrázku 6.20.



Obr. 6.20 Diagram tříd komponenty převodu akce

Konfigurační pravidla pro test převodu jsou zapsány v tomto tvaru.

```
<?php
$configuration = [
    "default,submit" => "view",
    "default,toggle*" => "edit",
    "*,submit" => "edit",
    "default," => "default-without-signal",
    "default,*" => "default-with-signal",
    "*,*" => "edit"
];
```

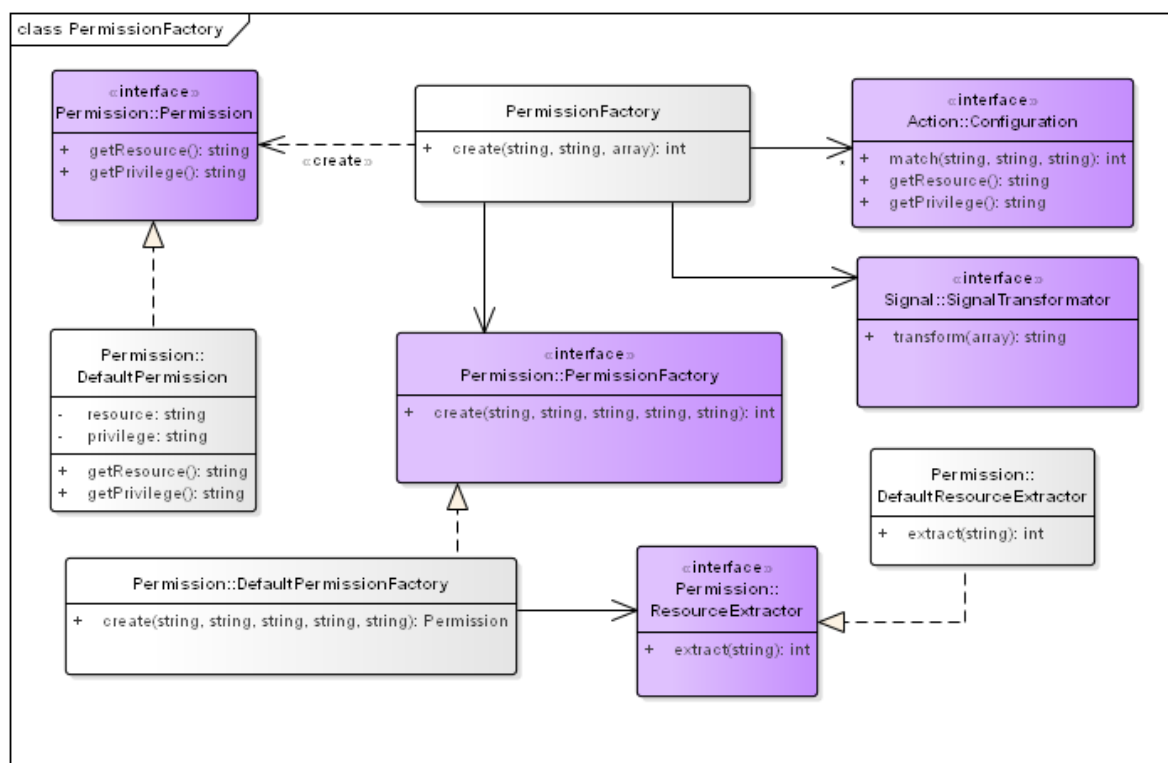
Testovací pravidla převodu jsou.

```

<?php
Assert::same('edit', $transformator->transform('default', 'toggleactive'));
Assert::same('view', $transformator->transform('default', 'submit'));
Assert::same('edit', $transformator->transform('edit', 'submit'));
Assert::same('edit', $transformator->transform('add', ''));
Assert::same('default-with-signal', $transformator->transform('default', 'some'));
Assert::same('default-without-signal', $transformator->transform('default', ''));

```

Diagram tříd systému pro převod adresy složeného z dílčích komponent je uveden na obrázku 6.21.



Obr. 6.21 Diagram tříd systému převodu presenteru

## 6.11 Využití cache

Požadavek zadání stanovuje využití paměti cache pro realizaci nefunkčního požadavku řízení zátěže hostujícího serveru. Paměť cache má smysl v aplikaci využít pouze v místech, kde bude dosaženo vhodného poměru počtu úspěšných a celkových dotazů do cache. Typicky se bude jednat o často navštěvovaná místa aplikace (stránky) kde obsah nebude velmi často měněn. Před spuštěním aplikace nejsou k dispozici údaje o vytížení jednotlivých částí aplikace. Stanovení míst kde je vhodné paměť cache implementovat bude provedena na základě zkušenosti z obdobných projektů. Návrh všech modulů aplikace však musí počítat s možností dodatečné implementace cache.

Obsah paměti cache musí být vhodným způsobem řízen. Po změně dat v aplikaci musí být zajištěno zneplatnění obsahu paměti cache. Obsah paměti cache u kterého je přípustné aby ke zneplatnění došlo v rozmezí stanoveného maximálního časového úseku bude ke zneplatnění využito nastavení času expirace obsahu. Obsah paměti cache který musí být zneplatněn neprodleně po změně obsahu databáze bude řízen procesem zneplatnění obsahu.

Proces řízení obsahu paměti cache bude využívat systému pro zneplatnění obsahu, systému pro značení obsahu a ze spouštěcího systému.

Systém pro značení obsahu bude realizován třídou obsahující statické metody. Metody budou vracet textový řetězec, který bude použit jako identifikátor obsahu. Například článek s ID 26 může být v cache identifikován jako řetězec *article/26*. Návrh třídy je uveden níže.

```
<?php
class Tagger
{
    /**
     * @param int $id
     * @return string
     */
    public static function article($id)
    {
        return "article/$id";
    }
}
```

Systém pro zneplatnění obsahu bude realizován třídou, jejíž metody zajistí provedení zneplatnění příslušného obsahu paměti. Parametry metod budou ID záznamů. Návrh rozhraní třídy je uveden na příkladu.

```
<?php
class Invalidator
{
    /**
     * @param int $id
     * @return void
     */
    public function article($id) {
        $tag = Tagger::article($id);
        $this->invalidate($tag);
    }
}
```

Proces spouštění procesu invalidace bude řízen pomocí událostí, které jsou realizovány pomocí vlastnosti přídy Object frameworku Nette a systému zpracování událostí *kdyby/events*. Vyvolání události je uvedeno příkladem.

```
<?php
class ConcretePresenter extends Presenter
{
    /**
     * @var callable[]
     */
    public $onDataChange = [];

    public function actionDelete($id) {
        $this->deleteRecord($id);
        $this->onDataChange($id); // trigger event
    }
}
```

## 7 IMPLEMENTACE

Tato kapitola popisuje klíčové momenty implementace jednotlivých komponent architektury. Implementace proběhla dle připraveného návrhu řešení, které bylo podrobně rozepsáno v předchozí kapitole.

### 7.1 Framework Nette na platformě PHP

Pro zajištění řízené správy verzí interních i externích knihoven pomocí nástroje Composer je nutné, aby součástí projektu byl verzován i soubor `composer.lock`, který obsahuje informace o aktuálně nainstalovaných knihovnách. Cílem je, aby byla vždy prováděna pouze operace `install` místo operace `update`. Blíže je tato problematika popsána v teoretické části.

#### 7.1.1 Režim maintenance

Řízení režimu maintenance bude spočívat v přejmenování souboru `maintenance.php` umístěného v adresáři se vstupním bodem aplikace. Soubor `maintenance.php` obsahuje kód pro zobrazení informací o nedostupnosti aplikace. Vstupní bod (`index.php`) obsahuje kód, který pokud existuje soubor `maintenance.php`, zajistí jeho spuštění a skončí.

```
<?php
if (is_file(__DIR__ . '/maintenance.php')) {
    include 'maintenance.php';
    exit;
}
```

Pro řízení režimu maintenance jsou k dispozici dva shellové skripty. Skript `maintenance_on.sh` režim maintenance zapíná a skript `maintenance_off.sh` vypíná. Obsah těchto skriptů je velmi podobný.

```
#!/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )";
PATH="$DIR/www";
MV=/bin/mv;
filenamefrom="$PATH/.maintenance.php";
filenameto="$PATH/maintenance.php";
if [ -f "$filenamefrom" ]
then
    $MV "$filenamefrom" "$filenameto";
fi
```

### 7.1.2 Vymazání cache

Paměť cache je spravovaná skrze rozhraní *IStorage*, které je implementováno v balíčku *nette/caching*. Vymazání obsahu celého uložště lze provést voláním funkce *IStorage::clean*. Implementace této metody v uložšti Redis ovšem trpí tím, že nejprve jsou získány všechny perzistované klíče a poté je pro každý klíč provedeno mazání. Výhodnější pro implementaci pomocí Redis je volat přímo metodu klienta *RedisClient::flushDb()*. Tento způsob je daleko výhodnější, neboť spočívá pouze v jednom příkazu zaslanému serveru Redis. Skript pro mazání obsahu cache *cleanCache.php* má tuto podobu.

```
#!/usr/bin/env php
<?php
$configurator = require_once(__DIR__ . '/app/bootstrap.php');
$storage = $configurator->getByType('Nette\Caching\IStorage', false);

if ($storage instanceof \Kdyby\Redis\RedisStorage) {
    $client = $configurator->getByType('Kdyby\Redis\RedisClient', false);
    $client->flushDb();
} else {
    $storage->clean([\Nette\Caching\Cache::ALL]);
}
```

### 7.1.3 Vymazání obsahu adresáře temp

Adresář temp slouží jako uložště pro kompilaci DI kontejneru, šablon a jako uložště pro další dočasné soubory. Při aktualizaci aplikaci na jiném než vývojovém serveru nedochází ke sledování změn ve struktuře aplikace a je nutné překompilované soubory vymazat tak, aby se aktuální verze vygenerovaly s prvním požadavkem. Vymazání obsahu bude zajištěno jednoduchým skriptem *cleanTemp.php*.

```
#!/usr/bin/env php
<?php
$rootDir = __DIR__;

function rmdir($dir) {
    if (is_dir($dir)) {
        $objects = scandir($dir);
        foreach ($objects as $object) {
            if ($object != "." && $object != "..") {
                if (filetype($dir . "/" . $object) == "dir") {
                    rmdir($dir . "/" . $object);
                } else {
                    unlink($dir . "/" . $object);
                }
            }
        }
    }
}
```

```
    }
  }
}
reset($objects);
rmdir($dir);
}
}

rmdir($rootDir . '/temp/cache/');
```

#### 7.1.4 Implementace nástroje pro aktualizaci databáze

Implementace nástroje pro aktualizaci struktury databáze podle návrhu proběhla formou privátního balíčku *esports/database-update*. Pro instalaci balíčku se musí v konfiguračním souboru aplikace nástroje Composer specifikovat uložení a zajistit potřebná oprávnění pro přístup. Balíček obsahuje i implementaci UI pro vizuální kontrolu připravených skriptů. Implementace UI využívá pro výpis formou datagridu balíček *o5/grido* a jeho komponentu *Grid*. Implementace UI vyžadovala rozšíření metod třídy *Updater* sloužící pro získání informací o vnitřním stavu procesu aktualizace. Balíček obsahuje trait *IsDBUpdaterPresenter*, který lze využít v libovolném presenteru v infrastruktuře UI administrace. Tento trait integruje správce aktualizací do aplikace. Následující příklad ilustruje způsob použití.

```
<?php
class DBUpdaterPresenter extends BasePresenter {

    use IsDBUpdaterPresenter;

    /**
     * @var ScriptTableFactory
     * @inject
     */
    public $scriptTableFactory;

    /**
     * @return ScriptTable
     */
    protected function createComponentScriptTable() {
        return $this->scriptTableFactory->create();
    }
}
}
```

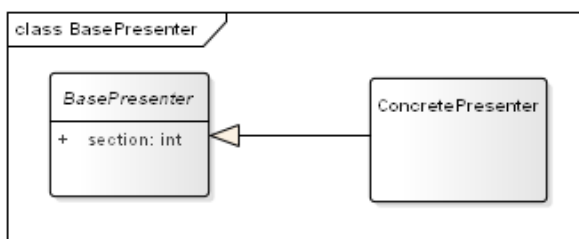
Implementace nástroje pro aktualizaci obsahuje pro zjednodušení začlenění do apli-

kace rozšíření DI kontejneru *DBUpdaterExtension*. Rozšíření *DBUpdaterExtension* umožňuje snadnou konfiguraci umístění adresáře s SQL skripty a také specifikovat název odkládací tabulky. Příklad konfigurace rozšíření je uveden níže.

```
extensions:  
  dbupdater: Esports\DBUpdater\DBUpdaterExtension  
  
dbupdater:  
  path: '%appDir%/../db/scripts'  
  table: version_db
```

## 7.2 Tématické rozdělení obsahu

Implementace tématického rozdělení obsahu s využitím modulů a sekcí vyžaduje, aby jakýkoliv presenter v infrastruktuře UI měl přístup k identifikátoru aktuálně zobrazené sekce. Důvodem je, že jeden modul může zobrazovat obsah pro více sekcí. Identifikátor sekce musí být automaticky předán jako parametr při tvorbě odkazů aplikace. Implementace proto využívá vlastnosti perzistentích parametrů Nette. Tyto parametry jsou automaticky předávány do procesu tvorby odkazů, nemusí se tento parametr explicitně uvádět. Aby se perzistentní parametr vztahoval na všechny presentery infrastruktury UI, musí být tento parametr definován ve třídě určené jako společný předek všem presenterům. Model pro tuto situaci je v diagramu tříd na obrázku 7.1.



Obr. 7.1 Diagram generalizace presenteru

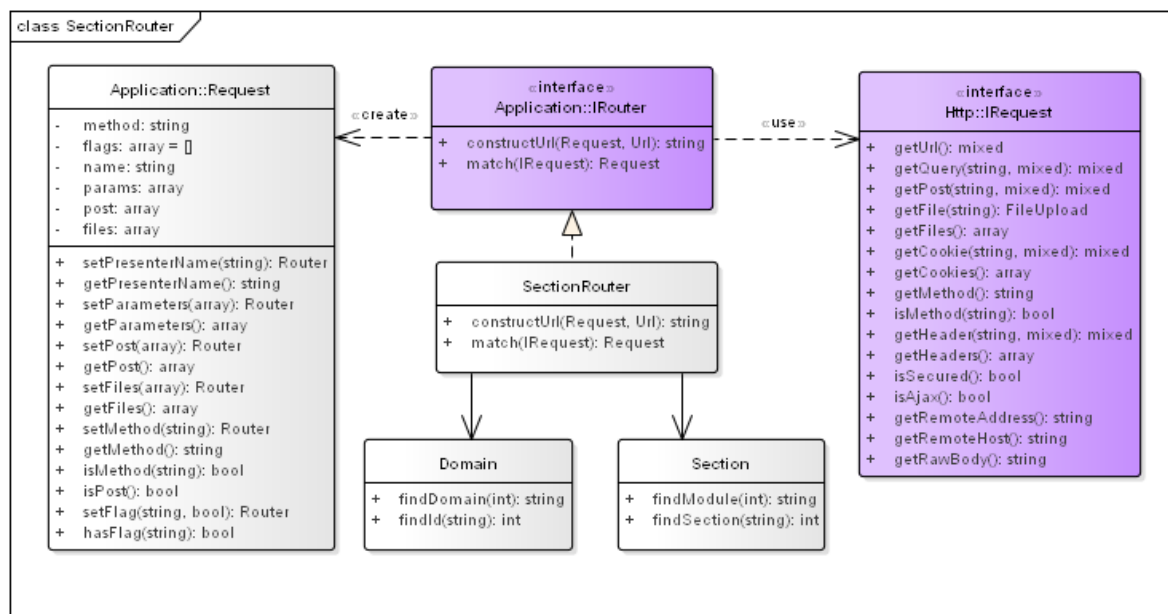
Řešení pomocí předávání parametru sekce bude klást požadavek na Router řídicí tvorbu odkazů. Požadavek bude spočívat ve vhodném zakomponování identifikátoru sekce do struktury modelu URL.

## 7.3 Strukturované URL

Realizace požadavku na vytvoření routovače zajišťujícího tvorby a překladu strukturovaného tvaru URL je popsána diagramem tříd uvedeném na obrázku 7.2.

Strukturovaného URL je dosaženo implementací rozhraní *IRouter*. Implementace je Továrna realizovaná třídou *RouterFactory* vrací instanci třídy implementující rozhraní *IRouter*, které aplikace využívá k překladu adres. *RouterFactory* využívá tříd





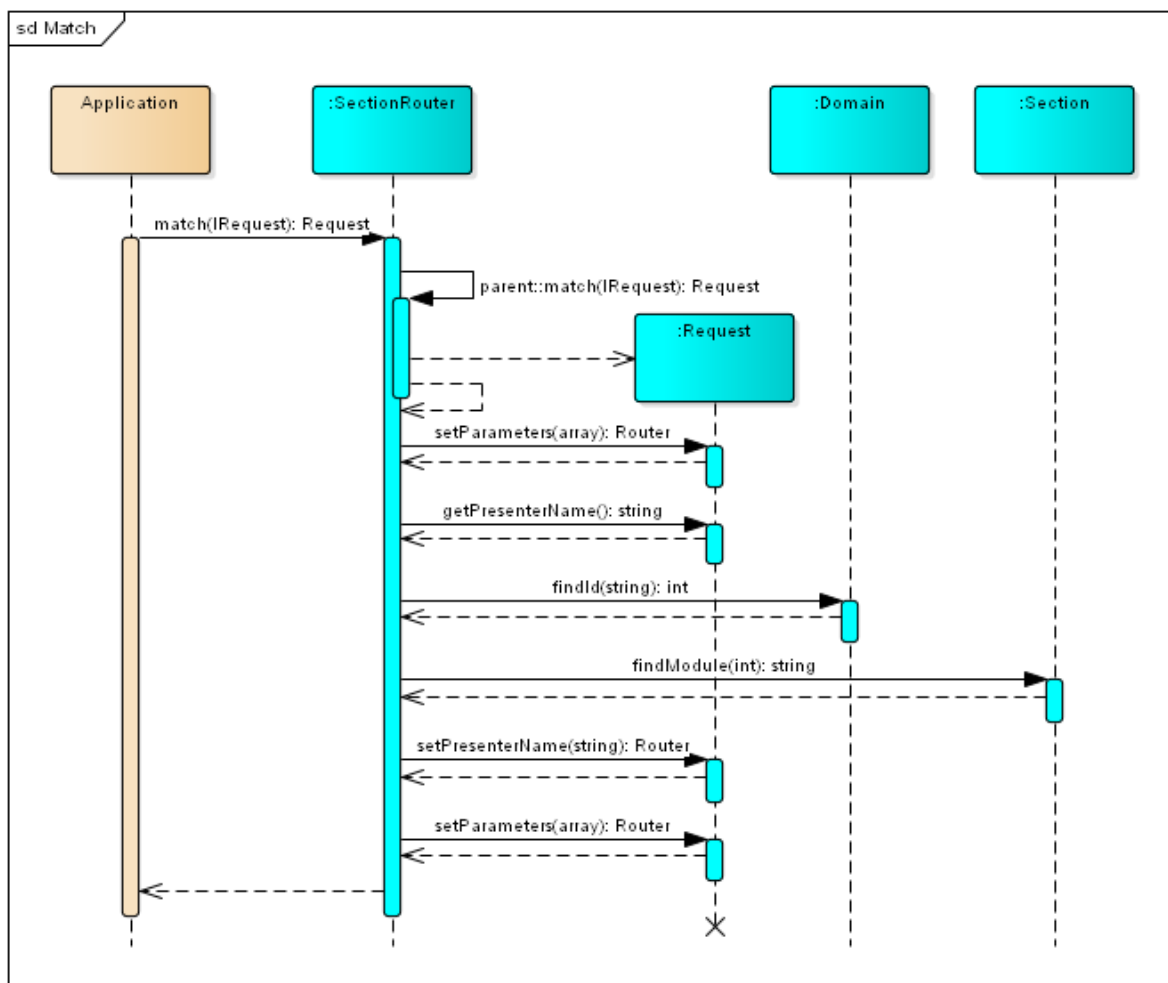
Obr. 7.3 Diagram tříd realizace třídy SectionRouter

pravidla za využití továrny *SectionRouterFactory* vracející novou instanci třídy *SectionRouter*.

```

<?php
$mask = "//[<section>.]<domain>
↳ [^.*]\.[^./]*>/<module>/<presenter>/<action>[/<id>]";
$metadata = [
    'section' => '',
    'module' => [
        Route::VALUE => 'Dashboard',
        Route::FILTER_TABLE => [
            'clanky' => 'Article',
            'zapasy' => 'Program',
            'tym' => 'Team',
        ]
    ],
    'presenter' => [
        Route::VALUE => 'Homepage',
        Route::FILTER_TABLE => [
            'archiv-zprav' => 'Article',
            'hraci' => 'Player',
            'historicke-statistiky' => 'HistoricalStats',
        ]
    ],
    'action' => 'default',
    'id' => NULL
];

```



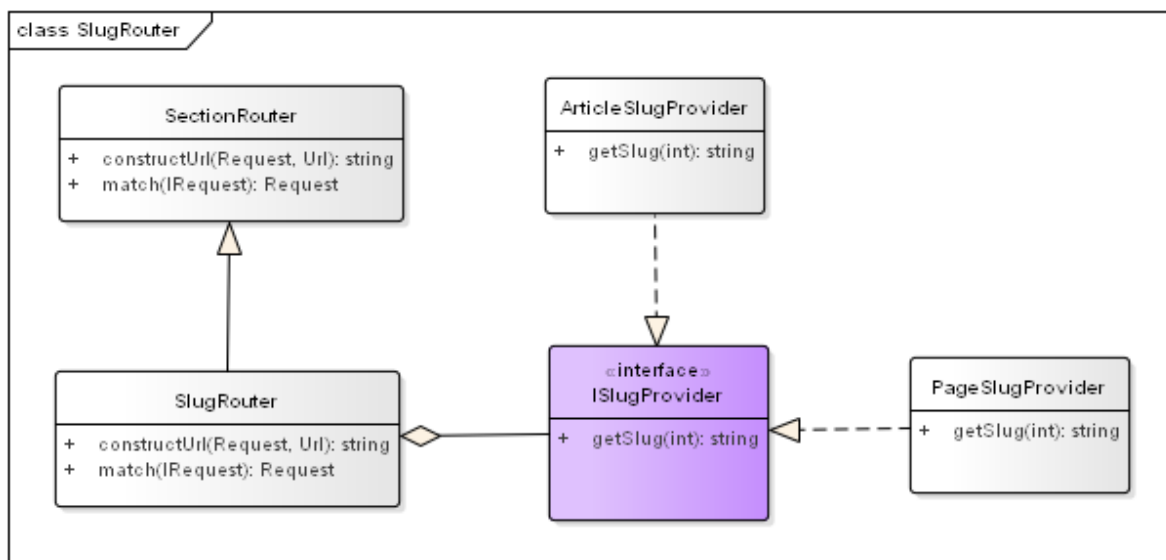
Obr. 7.4 Sekvenční diagram znázorňující převod domény

```
$router = $sectionRouterFactory->create($mask, $metadata);
```

#### 7.4 Databázová vrstva nette/database

Reimplementace balíčku *nette/database* je provedena klonováním (operace fork) původního repozitáře umístěného na serveru GitHub. Touto cestou je možné postupem času začleňovat změny v původním repozitáři. K nahrazení balíčků jinou implementací tak, aby v systému zůstala zachována reference na původní, slouží v systému Composer direktiva `replace`. Verze nahrazeného balíčku se většinou uvádí jako verze, ze které proběhlo klonování.

```
{
  "replace": {
    "nette/database": "2.3.0"
  }
}
```



Obr. 7.5 Diagram tříd realizace třídy SlugRouter

Testy definované ve fázi návrhu poskytují dostatečné pokrytí vytvářené funkcionality. Začleněním nových testů mezi původní testy balíčku a ověřením jejich neprůchodnosti, bylo dosaženo výchozího stavu pro implementaci. Stěžejní místo implementace nové funkcionality představuje třída *SqlBuilder*. Třída *SqlBuilder* je zodpovědná za tvorbu výsledného SQL dotazu na základě parametrů, které jsou jí předávány prostřednictvím třídy *Selection*.

V prvním kroku byly do třídy *Selection* přidány funkce definované v testu. Nová metoda *Selection::left* je strukturou totožná s metodou *Selection::where*. Rozdíl mezi *left* a *where* spočívá pouze ve volání rozdílné metody *SqlBuilderu*. Aby nedošlo k duplikaci kódu, byla metoda *where* refaktorována. Refaktor spočíval ve vyčlenění společné části kódu do nové metody *condition* a nahrazením obsahu metody *where*. Refaktorovaná část kódu je uvedena níže.

```

<?php
/**
 * Adds condition, more calls appends with AND.
 * @param string method name [where/left]
 * @param string condition possibly containing ?
 * @param mixed
 * @param mixed ...
 * @return self
 * @internal
 */
public function condition($method, $condition, $parameters = array())
{
    if (is_array($condition) && $parameters === array()) { // where(array('column1' =>
        ↪ 1, 'column2 > ?' => 2))
  
```

```
        foreach ($condition as $key => $val) {
            if (is_int($key)) {
                $this->$method($val); // where('full condition')
            } else {
                $this->$method($key, $val); // where('column', 1)
            }
        }
        return $this;
    }

    $this->emptyResultSet();
    $args = func_get_args();
    array_shift($args);
    call_user_func_array(array($this->sqlBuilder, "add" . ucfirst($method)), $args);
    return $this;
}

/**
 * Adds where condition, more calls appends with AND.
 * @param string condition possibly containing ?
 * @param mixed
 * @param mixed ...
 * @return self
 */
public function where($condition, $parameters = array())
{
    $args = func_get_args();
    array_unshift($args, 'where');
    return call_user_func_array($this->condition, $args);
}
```

Následující krok implementace spočíval v definici testů pro *SqlBuilder*. Tyto testy odrážejí funkcionality navrženou pomocí testů stanovených v návrhu. Implementace podmínek klauzule ON je velmi podobná implementaci podmínky WHERE, liší se pouze umístěním ve výsledném SQL. Ve vnitřní struktuře *SqlBuilderu* implementací metody *left* nedojde k žádnému nechtěnému narušení funkcionality. Obdobně i funkce *alias* nenaruší funkce *SqlBuilderu*. Předpoklad byl nakonec ověřen původními testy. V prvním kroku implementace nových funkcí do *SqlBuilderu* byla refaktorována metoda *where* obdobným způsobem jako ve třídě *Selection*. Funkce *where* a *left* se zde vnitřně liší pouze hodnotou klíče asociativního pole, ve kterém jsou uloženy zpracované parametry.

Místem navázání funkcionality *left* je metoda *SqlBuilder::buildSelectQuery*. Tato metoda zajišťuje sestaví dotazu typu SELECT. Navázání spočívalo ve volání metody

*SqlBuilder::buildQueryJoins*, jejíž parametry byly rozšířeny nový parametr obsahující zpracované podmínky pro klauzuli ON. Klíčová části začlenění kódu jsou uvedeny v zde.

```
<?php
$leftConditions = $this->left;
foreach($leftConditions as &$leftCondition){
    $this->parseJoins($joins, $leftCondition);
}
$queryJoins = $this->buildQueryJoins($joins,
    ↪ $this->buildLeftJoinConditions($leftConditions));
```

O vytvoření hodnoty pro nový parametr metody *SqlBuilder::buildQueryJoins* je zodpovědná metoda *SqlBuilder::buildLeftJoinConditions*. Tato metoda provádí postupnou iteraci skrze předanou kolekci pravidel a pokud nalezne odpovídající pravidlo, doplní příslušnou část SQL dotazu do výstupního pole. Změna v metodě *SqlBuilder::buildQueryJoins* spočívá pouze v kontrole, zda pro zpracováváný příkaz existují podmínky ve vstupním poli a v případě nalezení je začlenění do SQL dotazu. Výsledek je uvedený v následujícím příkladu.

```
<?php
protected function buildQueryJoins(array $joins, $leftConditions = array())
{
    $return = '';
    foreach ($joins as $join) {
        list($joinTable, $joinAlias, $table, $tableColumn, $joinColumn) = $join;

        $additionalConditions = '';
        if (isset($leftConditions[$joinAlias]) && count($leftConditions[$joinAlias])) {
            $additionalConditions = ' AND (' . $leftConditions[$joinAlias] . ')';
        }

        $return .=
            " LEFT JOIN {$joinTable}" . ($joinTable !== $joinAlias ? " {$joinAlias}" : '')
    ↪ .
            " ON {$table}.{$tableColumn} =
    ↪ {$joinAlias}.{$joinColumn}{$additionalConditions}";
    }
    return $return;
}

protected function buildLeftJoinConditions($allLeftJoinConditions)
{
    $conditions = array();
    foreach ($allLeftJoinConditions as $condition) {
```

```
if (strpos($condition, '.') === false) {
    continue;
}
$condition = Strings::trim($condition);
$table = Strings::replace($condition, '~\..*$~');
$table = Strings::replace($table, '~^.* ~');
$table = Strings::replace($table, '~^.*\(~');
if (!isset($conditions[$table])) {
    $conditions[$table] = $condition;
} else {
    $conditions[$table] .= " AND " . $condition;
}
}
return $conditions;
}
```

Těžiště začlenění funkcionality tvorby aliasů pomocí funkce *alias*, spočívá v přidání nové podmínky v metodě *SqlBuilder::parseJoinsCb*. Zde se iteruje přes nalezené řetězce symbolizující názvy tabulek, které je nutné připojit formou LEFT JOIN do výsledného SQL dotazu. Podmínka je uvedena na příkladu.

```
<?php
if (isset($this->aliases[$keyMatch['key']])) {
    $aliasJoin = $this->parseAlias($joins, $keyMatch['key'], $keyMatch['del']);
    list($table, $parentAlias, $column, $primary) = $aliasJoin;
}
```

Funkce *SqlAlias::parseAlias* na kterou se odkazuje výše uvedený kód slouží k rozparsování zadaného aliasu a přidání záznamů o připojených tabulkách v případě, že je alias vyžaduje. Pro správné zpracování joinů metodou *SqlBuilder::parseJoins* je nutné doplnit aliasovanou tabulku o libovolný atribut. Název atributu nehraje význam, neboť po zpracování aliasů je vyřazen. Výsledná metoda je uvedena na příkladu.

```
<?php
protected function parseAlias(& $joins, $aliasKey, $aliasDelimiter)
{
    if ($aliasDelimiter !== '.') {
        throw new Nette\InvalidArgumentException("Bad syntax when using alias. There
        ↪ cannot be ':$aliasKey...', must be '$aliasKey...");
    }
    $query = $aliasDelimiter . $this->aliases[$aliasKey] . ".x";
    $tmp = array();
    $this->parseJoins($tmp, $query);
    $aliasJoin = end($tmp);
}
```

```

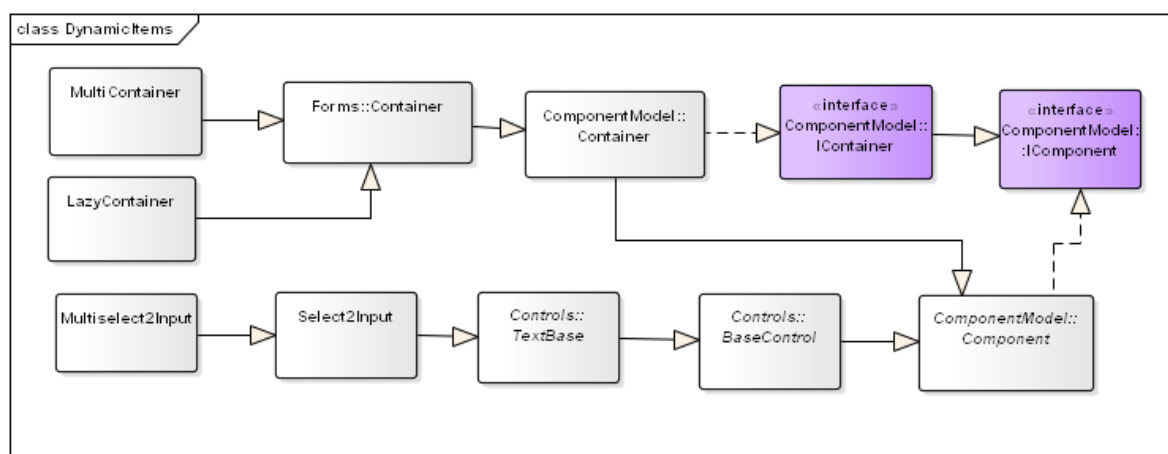
array_pop($tmp);
foreach ($tmp as $key => $join) {
    $joins[$key] = $join;
}
return $aliasJoin;
}

```

Reimplementovaný balíček *nette/database* je distribuován formou privátního uložiště s názvem *esports/database*. Projekt využívající tento balíček musí specifikovat uložisko v konfiguraci nástroje Composer a rovněž musí být zajištěno přidělení oprávnění přístupu. Životní cyklus správy balíčku bude obsahovat začlenění obsahu po vydání nové verze *nette/database*.

## 7.5 Dynamické formuláře

Implementace dynamických formulářů proběhla na základě navrženého chování a testů. Diagram tříd je uveden na obrázku 7.6. Výsledkem je privátní balíček *esports/forms-extension*. Pro využití balíčku v projektu je nutné nastavit uložisko a zajistit oprávnění pro přístup.



Obr. 7.6 Diagram tříd implementace dynamických prvků formuláře

Kontejner *LazyContainer* implementující funkci dynamické změny obsahu kontejneru vyžaduje při použití dodržet specifický postup. Postup spočívá ve volání metody kontejneru *LazyContainer::setDefault* při plnění výchozích hodnot formuláře, místo zápisu hodnoty přímo do komponenty uvnitř kontejneru. Tento postup je nutný, neboť implementace modelu komponent frameworku Nette disponuje vlastností lazy - komponenty jsou vytvářeny až tehdy, kdy o ně někdo požádá. Vzhledem k vnitřní struktuře kontejneru je spuštění předávané funkce pro vytvoření obsahu kontejneru spouštěno v metodě *LazyContainer::getComponents*. Tato metoda je ale volána v průběhu životního cyklu vícekrát. Z tohoto důvodu je kontrolováno zda byl kontejner již inicializován a

pokud ano, nová inicializace neproběhne. První inicializace však proběhne již při vkládání kontejneru do stromu komponent. Jelikož funkce vracející obsah kontejneru se může snažit získat hodnotu z komponenty formuláře, která ještě nebyla inicializována (typicky při zpracování odeslaného formuláře, kdy se nejprve formulář vytvoří a teprve pak se naplní hodnoty formuláře z dat odeslaných HTTP požadavkem). Z těchto důvodů metoda *LazyContainer::setValues* provádí reinicializaci kontejneru. Metoda *LazyContainer::setDefault*s v případě neodeslaného formuláře zajistí reinicializaci, neboť vnitřně volá metody *LazyContainer::setValues*.

Součástí balíčku *esports/forms-extension* je i rozšíření DI kontejneru pro snadnou registraci nových komponent formuláře. Rozšíření DI využívá metody *Object::addExtensionMethod* pro registraci funkcí do implementace formulářů balíčku *nette/forms*. Registrace rozšíření se provádí v konfiguračním souboru aplikace.

```
extensions:  
  - Esports\Forms\FormsExtension
```

## 7.6 Flexibilní modelová vrstva

Implementace repozitářů odhalila skutečnost, že většina repozitářů obsahuje metody, které jsou pro repozitáře vždy stejné. Aby nedocházelo k duplikaci kódu, byly vytvořeny traity obsahující zmiňované metody. Traity byly zvoleny jako alternativa pro vytvoření abstraktního repozitáře, ze kterého by musely všechny repozitáře dědit.

Repozitář využívající společný kód má následující strukturu.

```
<?php  
use Nette\Database\Context;  
  
class MatchRepository  
{  
    use IsSimpleRepository;  
  
    public function __construct($tableName, Context $db)  
    {  
        $this->setTableName($tableName);  
    }  
}
```

Trait *IsSimpleRepository* využívá služeb traitů *HasTable* a *IsCRUD*. Trait *HasTable* implementuje funkce pro práci s vrstvou *Table*. Trait *IsCRUD* implementuje CRUD funkce.

```
<?php  
use Nette\Database\Table\Selection;
```

```
trait HasTable
{

    /** @var string */
    private $tableName = '';

    /**
     * @return Selection
     */
    protected function table()
    {
        return $this->db->table($this->tableName);
    }

    /**
     * @param string $tableName
     */
    public function setTableName($tableName)
    {
        $this->tableName = $tableName;
    }

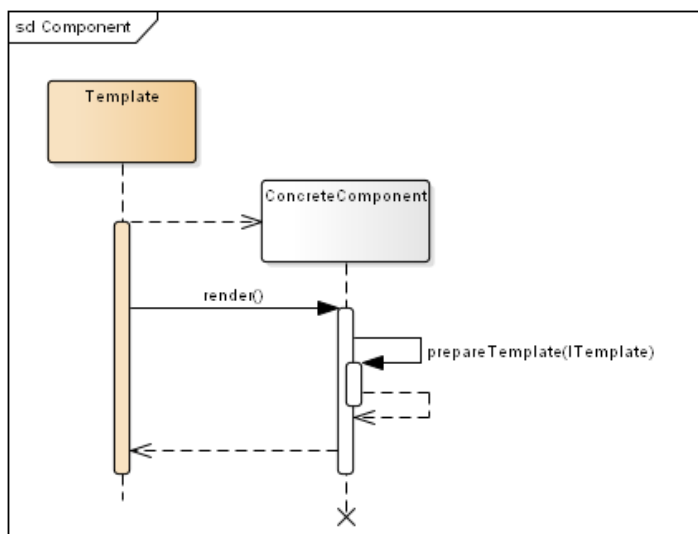
    /**
     * @return string
     */
    public function getTableName()
    {
        return $this->tableName;
    }

}
```

## 7.7 Znovupoužitelné komponenty uživatelského rozhraní

Všechny komponenty uživatelského rozhraní aplikace budou ve své implementaci dědit od třídy *Component*, která byla implementována na základě požadavků návrhu. Třída *Component* dědí od třídy *Control* frameworku Nette. Třída *Component* implementuje metody pro injektaci správce helperů a překladače textů. Životní cyklus jednoduché komponenty s kódem uvedeným níže je popsán sekvenčním diagramem na obrázku 7.7.

```
<?php
class DecisionList extends Component
{
    /**
```



Obr. 7.7 Sekvenční diagram vykreslení komponenty

```

* @var IDecisionProvider
*/
private $decisionProvider;

function __construct(IDecisionProvider $decisionProvider)
{
    parent::__construct();
    $this->decisionProvider = $decisionProvider;
}

protected function prepareTemplate(ITemplate $template)
{
    parent::prepareTemplate($template);
    $template->setFile(__DIR__ . '/decisionList.latte');
    $template->items = $this->decisionProvider->getDecision();
}
}

```

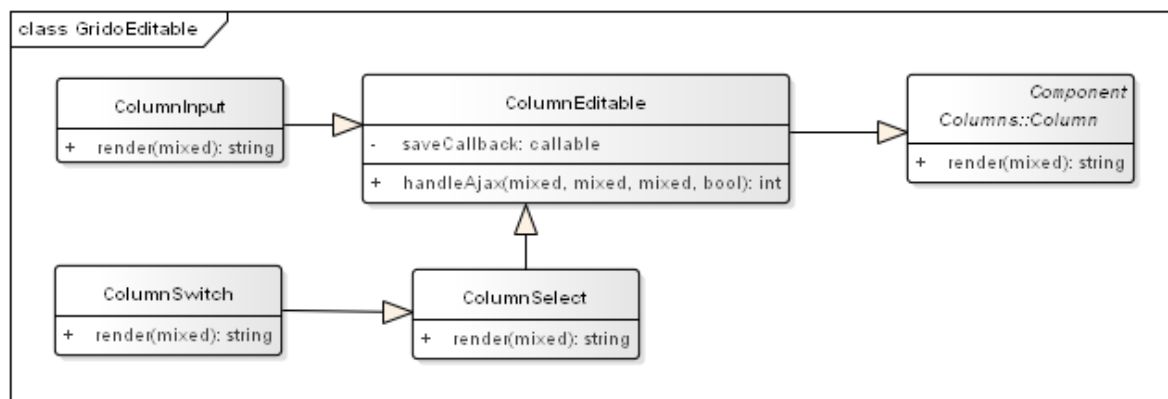
Registrace komponenty v konfiguračním souboru DI kontejneru vyžaduje explicitně specifikovat injektáž potřebných komponent. Na příkladu je znázorněna konfigurace pro komponentu použitou v předchozím příkladu.

```

services:
    component:
        class: DecisionList
        setup:
            - setTranslator()
            - setHelperLoader()
  
```

## 7.8 Inline editace

Realizace inline editace pro komponentu Grid proběhla formou privátního balíčku *esports/grido-editable-extension*. Implementace je zachycena v diagramu tříd na obrázku 7.8. Třídy *ColumnSwitch*, *ColumnSelect* a *ColumnInput* jsou zodpovědné za tvorbu interakčního UI prvku. Třída *ColumnEditable* je zodpovědná za zpracování události vyvolané změnou hodnoty prvku formou zpracování signálu.



Obr. 7.8 Diagram tříd implementace inline editace

Implementace rozhraní pro tvorbu sloupců odpovídající návrhu byla realizována využitím vlastnosti frameworku Nette dané metodou *Object::addExtensionMethod*. Pro usnadnění registrace je vytvořeno rozšíření DI kontejneru realizující registraci rozšiřujících metod. Konfigurace se v konfiguračním souboru aplikace zapisuje formou:

```

extensions:
  - Esports\Grido\GridoEditableExtension
  
```

## 7.9 Systém řízení uživatelských oprávnění

Implementace systému převodu uživatelských oprávnění proběhla formou privátního balíčku *esports/privileges* a je nutné v konfiguračním souboru aplikace nástroje Composer specifikovat uložště a zajistit oprávnění pro přístup do toho uložště.

Výsledná struktura odpovídá návrhu. Navíc je doplněna o utilitu *RightChecker*, která usnadňuje použití překladače adres v aplikaci. Oproti návrhu byly také rozšířeny testy. Součástí implementace je i rozšíření pro DI kontejner *PrivilegesExtension*. Rozšíření DI kontejneru umožňuje snadnou konfiguraci systému převodu pomocí parametrů zapsaných do konfiguračního souboru. Konfigurace obsahuje nejen definici konfiguračních pravidel převodu, ale umožňuje i vyměnit implementaci jednotlivých komponent zajišťující převod. Díky této konfiguraci lze jednoduše vyměnit výchozí implementaci za vlastní. Příklad konfiguračního souboru je uveden níže.

```

extensions:
  privileges: Esports\Privileges\PrivilegesExtension

privileges:
  rights:
    "Admin:User:User:" : "User:"
    "Admin:User:Role:" : "Role:"
    "Admin:Export:*:" : "Export:"
    "Admin:SubjectManager:DocumentFolder:" : "SubjectDocument:"
    "*:*" : "Default:"

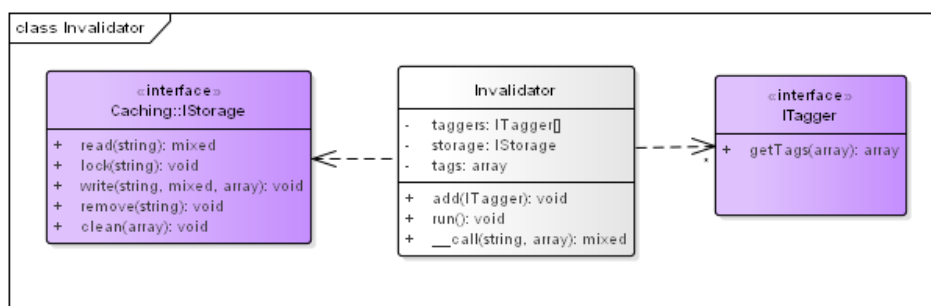
  privileges:
    "*,filter" : "view"
    "default,*" : "view"
    "*,*" : "edit"

  signal:
    "grid-form,submit" : "filter"
    "*-grid-form,submit" : "filter"
    "*,*" : "edit"

```

## 7.10 Využití cache

Implementace třídy realizující invalidaci cache je popsána diagramem tříd na obrázku 7.9.



Obr. 7.9 Diagram tříd realizace invalidátoru cache

Třída *Invalidator* obsahuje kolekci objektů implementujících rozhraní *ITagger*. Metoda rozhraní *ITagger::getTags* je určeno pro získání identifikátorů paměti cache. Návrátovou hodnotu tak tvoří vektor řetězců. Proces zneplatnění paměti je rozdělen do dvou fází. Fáze jsou odděleny z důvodu, aby zneplatnění paměti proběhlo v jednom kroku.

První fáze odpovídá návrhu. Je složena ve volání metod invalidátoru pro zneplatnění obsahu paměti. Voláním metod se však paměť nezneplatní okamžitě. Do privátního pole třídy *Invalidator* se pouze uloží tagy vrácené rozhraním *ITagger*.

Druhá fáze realizuje proces zneplatnění obsahu cache pomocí metody *IStorage::clean*. Proces je spuštěn voláním metody *Invalidator::run* přímo či v destrukturu třídy *Invalidator*. Jako parametr metody *IStorage::clean* je předán obsah vnitřního pole obsahující tagy vytvořené v první fázi. Po zneplatnění obsahu se obsah vnitřního pole třídy *Invalidator* vyprázdní.

Pro zpracování událostí oznamujících požadavek na zneplatnění obsahu, které jsou generovány v presenterech, slouží třída *CacheInvalidatorSubscriber*. Třída implementuje rozhraní *Subscriber* balíčku *kdyby/events*. Tato třída je zodpovědná za mapování přicházející události na místo zpracování. Zpracování události je tvořeno voláním metody třídy *Invalidator*. Ukázka invalidátoru je uvedena v následujícím příkladu.

```
<?php
use Kdyby\Events\Subscriber;
class CacheInvalidatorListener implements Subscriber
{

    /** @var Invalidator */
    private $invalidator;

    public function getSubscribedEvents()
    {
        return array(
            '\Admin\Article\ArticlePresenter::onDataChange' => 'article'
        );
    }

    public function article($id)
    {
        $this->invalidator->article($id);
    }
}
```

## 8 ZHODNOCENÍ PROVOZU

Zhodnocení je důležitým krokem pro získání zpětné vazby o realizaci projektu. Získané pozitivní i negativní zkušenosti se odrazí ve firemním known-how a umožní tyto zkušenosti promítnout do realizace dalších projektů, kde pomohou při klíčovém rozhodování. Tato kapitola popisuje krátké zhodnocení portálu po necelém půl roce provozu.

### 8.1 Udržovatelnost aplikace

Během prvních dvou měsíců provozu došlo v projektu k celé řadě změn. Iniciátorem těchto změn byly většinou připomínky návštěvníků portálu a také členů svazu. Velká část připomínek byla cílena na zpřehlednění či přeskupení bloků v grafickém rozhraní aplikace. Některé připomínky ovšem přinesly nutnost provést větší refaktoring modelové vrstvy a následně i vyšších vrstev aplikace. Zpracování požadavků a jejich začlenění do systému, se díky pružnému návrhu ukázalo jako bezproblémové. Dokumentace architektury provedená formou UML diagramů včetně zachycení interních procesů se ukázala jako klíčová a bezpodmínečně nutná pro bezproblémový chod a údržbu projektu. Úsilí věnované podrobné přípravě a realizaci architektury se tak mnohonásobně vrátilo.

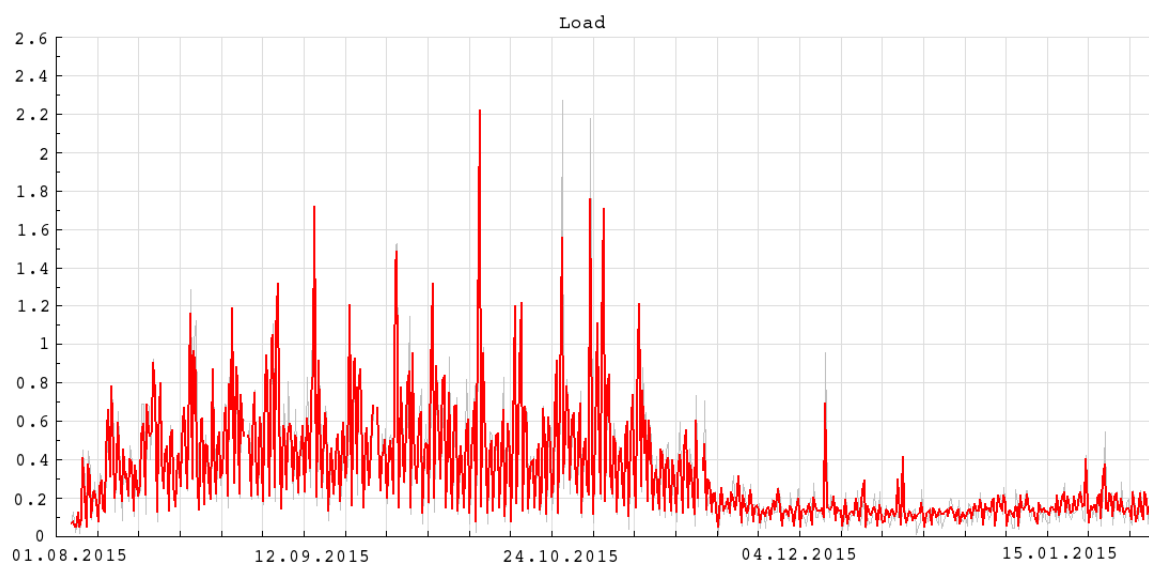
K rychlému a bezproblémovému zvládnutí zpracování připomínek na uživatelské rozhraní mohlo dojít díky pružnému návrhu prezentační vrstvy. Využití členění do modulů a vizuálních komponent zde sehrálo klíčovou roli stejně jako využití abstraktních datových zdrojů. Při realizaci připomínek se ve většině případů pouze vyměnil datový zdroj či zaměnily vizuální komponenty. Strukturování prezentační vrstvy do modulů a komponent stanovilo jasný trend, který bude v realizaci dalšího projektu určitě využit.

### 8.2 Automatizovaný proces aktualizace

Automatizovaný proces aktualizace zdrojových kódů aplikace včetně aktualizace struktury databáze sehrál při údržbě aplikace důležitou roli. Díky vhodně nastavenému procesu nedochází při začleňování změn do produkce k žádným nechtěným chybám, které by zapříčinily výpadek. Automatizovaný proces je výhodný zejména pro vývojáře, neboť změny provedené v aplikaci se ihned propagují na vývojový, testovací a produkční server. Testovací server je primárně určen k prezentování výsledku po začlenění a k poslední fázi testování. Nasazení změn na produkční systém probíhá podle striktních pravidel. Aktualizace probíhá po odsouhlasení všech členů týmu pouze ve vybraných dnech a v konkrétních hodinách. Pokud by došlo k neočekávané situaci, je vývojový tým připraven ihned reagovat.

### 8.3 Zátěž serveru

Jedním z nefunkčních požadavků portálu byla nízká zátěž hostitelského systému. Na obrázku 8.1 je zobrazen graf zátěže systému za celé období provozu. Z grafu je patrné, že se úroveň systému daří udržovat vzhledem ke čtyřem CPU jádrům na rozumné hodnotě  $<3$  a to i při špičkách. Špičky jsou způsobeny skokovým nárůstem návštěvnosti, které se dějí pravidelně na začátku týdne, kdy návštěvníci portálu hledají aktuální výsledky po odehraných kolech soutěží.



Obr. 8.1 Graf zátěže systému

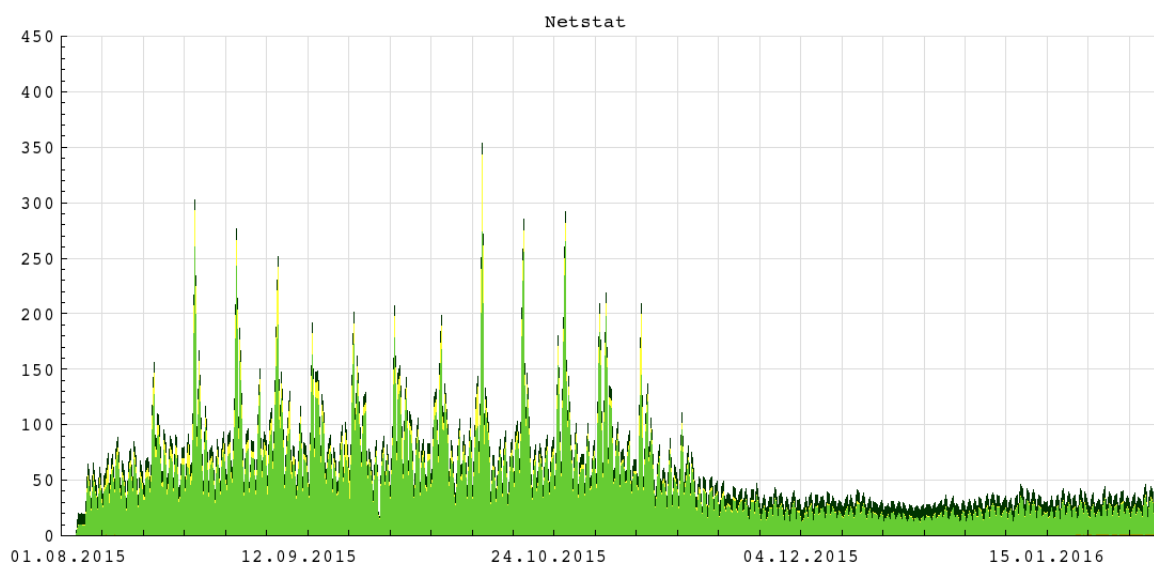
Z podrobného modelu zátěže serveru vypracované společností zajišťující hosting vyplývá, že nárůst zatížení je vzhledem k nárůstu připojení lineární. Z modelu byl stanoven přibližný maximální počet spojení a maximum dostupných prostředků. Dle těchto údajů bylo na serveru nastaveno omezení dostupnosti. Po překročení maximálních hodnot dojde k odmítnutí nového spojení. Pokud by spojení nebylo odmítnuto, mohlo by dojít k zahlcení serveru v důsledku přetížení. Graf síťových procesů za celé období provozu je uveden na obrázku 8.2.

### 8.4 Cache pomocí serveru Redis

Po nasazení aplikace do provozu byl pečlivě monitorován provoz serveru Redis. Důvodem monitorování byl sběr provozních údajů serveru. Z provozních údajů byl pravidelně vyhodnocován parametr *hit ratio*.

Zjištění parametrů probíhalo pomocí konzole Redisu *redis-cli*:

```
$ redis-cli info | grep keypace  
keyspace_hits:195754896
```



Obr. 8.2 Graf síťových spojení systému

```
keyspace_misses:29696598
```

Ze zjištěných hodnot bylo dopočítáno, že parametr *hit ratio* odpovídá cca 85%. Takto vysoká hodnota indikovala správné využívání paměti cache v aplikaci.

V průběhu vývoje aplikace bylo vytipováno ještě několik míst aplikace, kde bylo možné nasadit paměť cache. Po nasazení byla opět průběžně monitorována hodnota parametru *hit ratio*. Cílem bylo zjistit, zda provedená úprava byla efektivní, či nikoliv.

Celkově lze konstatovat, že implementace paměti cache s využitím serveru Redis je používána efektivně a velmi přispívá ke stabilitě a výkonu aplikace.

## 8.5 Kritické chyby

V průběhu druhého měsíce po spuštění byla v aplikaci identifikována kritická chyba. Tato chyba přímo nesouvisela s návrhem architektury a netýkala se uživatelského rozhraní ani administrace aplikace. Portál využívá externí datový zdroj pro získání výsledků domácích soutěží. Z tohoto zdroje se v pravidelných intervalech stahují aktualizované údaje. Následně se aktualizované údaje promítají v transakcích do interní databáze. Bohužel při návrhu importu byla podceněna konkurečnost přístupu do databáze. Importní SQL dotazy do relační databáze byly sestaveny tak, že využívaly více tabulek. Současně existovalo více druhů těchto SQL dotazů se stejnými tabulkami ale rozdílnou cílovou tabulkou zápisu. Důsledkem těchto rozdílných SQL dotazů prováděných paralelně byl deadlock. V důsledku deadlocku docházelo k přerušení importu a citelnému zpoždění aktualizace záznamů v databázi po změně ve zdrojovém systému. Čím více zpoždění import nabíral, tím častěji docházelo k deadlocku.

V následujícím příkladu jsou uvedeny dva SQL dotazy, u kterých docházelo při paralelním zpracování k deadlocku.

```
UPDATE 'match' m
  LEFT JOIN 'team_in_match' tmh ON tmh.match_id = m.id AND tmh.type = 'home'
  LEFT JOIN 'team_in_match' tmg ON tmg.match_id = m.id AND tmg.type = 'guest'
  SET
    m.match_status_code = 'finished',
    period_code = IF(tmh.penalty_goals > 0 OR tmg.penalty_goals > 0, 'ESO', 'ERT')
  WHERE m.guid IN (?)
```

```
UPDATE 'team_in_match' tm
  LEFT JOIN 'match' n ON m.id = tm.match_id
  SET
    goals=?, penalty_goals=?, match_guid=m.guid
  WHERE tm.team_guid IN (?)
```

V prvním SQL dotazu je aktualizována tabulka *match*. Aktualizace však vyžaduje připojení tabulky *team\_in\_match*, ze které se vybírají data potřebné pro aktualizaci. Druhý SQL dotaz aktualizuje tabulku *team\_in\_match*. Tento dotaz vyžaduje pro aktualizaci tabulku *match*. Vlivem paralelního provádění těchto dotazů došlo k situaci, kdy se v jednu chvíli pro první SQL zamknula tabulka *match* a pro druhé SQL tabulka *team\_in\_match*. Pro dokončení prvního SQL dotazu byla vyžadována tabulka *team\_in\_match*, která však byla zamknuta. Stejně tak druhý dotaz vyžadoval tabulku *match*, která tou dobou byla taky zamknuta. Server MySQL detekoval deadlock a vykonal vyvolání patřičné výjimky. Problém byl umocněn faktem, že v rámci jednoho SQL dotazu se prováděla aktualizace velkého množství dotazů. Provádění rozsáhlých SQL dotazů je náročné na čas, během kterého je tabulka zamknuta. Nevhodná konstrukce SQL dotazu spolu s dlouhou dobou zpracování umožnila častému vzniku deadlocku.

Situace byla vyřešena tak, že importní data byla rozdělena do malých bloků, které se následně zpracovávají sekvenčně. Dále se využily pro aktualizaci dat dočasné tabulky. Místo přímé aktualizace cílové tabulky se data nejprve vloží do dočasné tabulky. V druhém kroku se pak z dočasné tabulky aktualizuje cílová tabulka. Rozdělení aktualizace do dvou kroků a využití dočasné tabulky eliminuje možnost vzniku deadlocku.

Díky rychlému zachycení situace a realivně jednoduchému refactoru bylo rychle dosaženo nápravy. Z nastalé situace však vyplynulo velmi důležité ponaučení pro další realizace. Je důležité brát konkurečnost přístupu v úvahu a aktivně předcházet vzniku kritických sekcí.

## ZÁVĚR

Cílem této práce bylo navrhnout architekturu pro internetový portál fotbal.cz. V prvním kroku realizace návrhu architektury bylo nutné nejprve identifikovat všechny klíčové funkční a nefunkční požadavky aplikace. Identifikace požadavků proběhla na základě detailního rozboru všech požadavků zadavatele a požadavků projektového týmu. Jako klíčové požadavky byly vybrány ty, které zásadním způsobem ovlivňovaly podobu architektury aplikace. Tyto požadavky jsou uvedeny v kapitole věnované specifikaci požadavků.

Zpracování požadavků do podoby návrhu řešení je uvedeno v kapitole věnované návrhu řešení. Návrh řešení probíhal formou rozboru požadovaných funkcionalit, stanovení rozsahu konfigurace a volby technologií, které budou při realizaci využity. Navržené řešení je popsáno pomocí diagramů tříd a sekvenčních diagramů jazyka UML. Realizace vybraných požadavků počítá předem s výběrem konkrétní technologie, která je specifikována v zadání požadavku. Návrh řešení těchto vybraných požadavků proběhl formou identifikace možností dané technologie. Pro tuto technologii byly zkoumány schopnosti pokrytí stanovených požadavků. U nepokrytých požadavků byl proveden návrh řešení formou rozšíření funkcionality vybrané technologie.

Průběh realizace požadavků je popsán v kapitole věnované implementaci návrhu řešení. Jsou zde popsány klíčové fáze implementace pomocí diagramů jazyka UML, včetně příkladů konfigurace vytvořených komponent. Realizace vybraných požadavků byla provedena formou balíčku pro správce závislostí Composer. Tato forma realizace byla zvolena z důvodů snadné znovupoužitelnosti realizovaného řešení.

Podrobný rozbor požadavků, jejich zpracování do podoby návrhu řešení a následné implementace se ukázalo jako klíčové pro výsledný úspěch realizace projektu. V průběhu vývoje portálu došlo k několika zásadním změnám v požadavcích, které v důsledku znamenaly rozsáhlé změny funkcí aplikace. Vzhledem k podrobně zpracovanému návrhu architektury, jeho dokumentaci a testům, proběhlo začlenění nových požadavků velmi hladce. Tímto krokem bylo ověřeno splnění nefunkčního požadavku snadné modifikovatelnosti aplikace.

Po spuštění projektu do ostrého provozu bylo provedeno vyhodnocení provozní zátěže aplikačního i databázového serveru. Vyhodnocení získaných parametrů potvrdilo splnění nefunkčního požadavku nízké provozní zátěže. Vysoká hodnota hit ratio získaná ze serveru Redis ukázala, že implementace paměti cache je v aplikaci provedena správně a je umístěna ve vhodných částech aplikace.

V průběhu několika měsíců po spuštění projektu do ostrého provozu, byly realizovány požadavky na úpravu, rozšíření a přidání nových funkcí. Všechny tyto požadavky

mohly být bez obtíží začleněny a otestovány.

Vytyčený úkol, představující návrh architektury internetového portálu fotbal.cz, se podařilo úspěšně naplnit. Celkově lze návrh architektury hodnotit velmi pozitivně, neboť časová investice do podrobného návrhu zpracování se mnohonásobně vrátila díky možnostem vývoje a udržitelnosti aplikace. Navržená architektura aplikace také velmi výrazně přispěla k velmi dobré provozní stabilitě aplikace.

## SEZNAM POUŽITÉ LITERATURY

- [1] Introduction. *Composer* [online]. [cit. 2015-10-28]. Dostupné z: <https://getcomposer.org/doc/00-intro.md>
- [2] Repositories. *Composer* [online]. [cit. 2015-10-28]. Dostupné z: <https://getcomposer.org/doc/05-repositories.md>
- [3] Schema. *Composer* [online]. [cit. 2015-10-28]. Dostupné z: <https://getcomposer.org/doc/04-schema.md>
- [4] About Packagist. *Packagist* [online]. [cit. 2015-10-28]. Dostupné z: <https://packagist.org/about>
- [5] Redis. *Redis* [online]. [cit. 2015-10-28]. Dostupné z: <http://redis.io/>
- [6] ŠTRAUCH, Adam. Redis: key-value databáze v paměti i na disku. *Zdroják.cz* [online]. 2010 [cit. 2016-01-23]. Dostupné z: <https://www.zdrojak.cz/clanky/redis-key-value-database-v-pameti-i-na-disku/>
- [7] How fast is Redis? *Redis* [online]. [cit. 2015-10-28]. Dostupné z: <http://redis.io/topics/benchmarks>
- [8] About. *Git* [online]. [cit. 2015-10-28]. Dostupné z: <https://git-scm.com/about>
- [9] CHACON, Scott. *Pro Git*. Praha: CZ.NIC, c2009. CZ.NIC. ISBN 978-80-904248-1-4.
- [10] *GitHub* [online]. San Francisco (CA) [cit. 2015-10-28]. Dostupné z: <https://github.com>
- [11] Plans and pricing. *GitHub* [online]. [cit. 2015-10-28]. Dostupné z: <https://github.com/pricing>
- [12] *Bitbucket* [online]. San Francisco (CA) [cit. 2016-04-11]. Dostupné z: <https://bitbucket.org/>
- [13] Features. *Bitbucket* [online]. [cit. 2015-10-28]. Dostupné z: <https://bitbucket.org/product/features>
- [14] Pricing. *Bitbucket* [online]. [cit. 2015-10-28]. Dostupné z: <https://bitbucket.org/product/pricing>
- [15] How it all works. *Deploy* [online]. [cit. 2015-10-28]. Dostupné z: <https://www.deployhq.com/how>

- 
- [16] Pricing & Signup. *Deploy* [online]. [cit. 2015-10-28]. Dostupné z: <https://www.deployhq.com/packages>
- [17] Rychlý a pohodlný vývoj webových aplikací v PHP. *Nette Framework* [online]. [cit. 2015-10-28]. Dostupné z: <http://nette.org/cs/>
- [18] GRUDL, David. *NETTE REVOLUTION 2.2* [online]. [cit. 2015-10-28]. Dostupné z: <http://phpfashion.com/nette-revolution-2-2>
- [19] GRUDL, David. *Školení Nette, PHP, jQuery, React a JavaScriptu od Davida Grudla* [online]. [cit. 2015-10-28]. Dostupné z: <https://www.skoleniphp.cz/>
- [20] Download. *Nette Framework* [online]. [cit. 2015-10-28]. Dostupné z: <https://nette.org/cs/download>
- [21] Nette Foundation. *GitHub* [online]. [cit. 2015-10-28]. Dostupné z: <https://github.com/nette/>
- [22] Dependency Injection. *Nette Framework* [online]. [cit. 2015-10-28]. Dostupné z: <https://doc.nette.org/cs/2.3/dependency-injection>
- [23] MARTIN, Robert C, Michael C FEATHERS, Timothy R OTTINGER, Jeffrey J LANGR, Brett L SCHUCHERT, James W GRENNING a Kevin Dean WAMPLER. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River: Prentice-Hall, 2009. Robert C. Martin series. ISBN 978-0-13-235088-4.
- [24] GRUDL, David. *NABUŠENÉ DI SRDCE PRO VAŠE APLIKACE* [online]. 2015 [cit. 2015-10-28]. Dostupné z: <http://phpfashion.com/nabusene-di-srdce-pro-vase-aplikace>
- [25] MVC aplikace & presentery. *Nette Framework* [online]. [cit. 2015-10-28]. Dostupné z: <https://doc.nette.org/cs/2.3/presenters>
- [26] Přihlašování & oprávnění uživatelů. *Nette Framework* [online]. [cit. 2015-10-28]. Dostupné z: <https://doc.nette.org/cs/2.3/access-control>
- [27] Cache. *Nette Framework* [online]. [cit. 2015-10-28]. Dostupné z: <https://doc.nette.org/cs/2.3/caching>
- [28] Nette/caching: Cache layer with set of storages. *GitHub* [online]. [cit. 2015-10-28]. Dostupné z: <https://github.com/nette/caching>
- [29] Databáze. *Nette Framework* [online]. [cit. 2015-10-28]. Dostupné z: <https://doc.nette.org/cs/2.3/database>

- [30] Nette/database: Nette Database layer. *GitHub* [online]. [cit. 2015-10-28]. Dostupné z: <https://github.com/nette/database>
- [31] FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, c2003. Addison-Wesley signature series. ISBN 03-211-2742-0.
- [32] PROCHÁZKA, Filip. Kdyby/Redis: Redis storage for Nette Framework. *GitHub* [online]. [cit. 2015-10-28]. Dostupné z: <https://github.com/kdyby/redis>
- [33] PROCHÁZKA, Filip. Kdyby/Events: Events for Nette Framework. *GitHub* [online]. [cit. 2016-01-23]. Dostupné z: <https://github.com/kdyby/events>
- [34] BUGYÍK, Petr. Documentation: Grido DataGrid for Nette Framework!. *GitHub* [online]. [cit. 2016-01-23]. Dostupné z: <http://o5.github.io/grido-examples/documentation.cs.html>
- [35] BUGYÍK, Petr. *Home: Grido DataGrid for Nette Framework* [online]. [cit. 2016-01-23]. Dostupné z: <http://o5.github.io/grido-examples/>
- [36] BUGYÍK, Petr. O5/gridido: Grido - DataGrid for Nette Framework. *GitHub* [online]. [cit. 2016-01-23]. Dostupné z: <https://github.com/o5/gridido>
- [37] MYERSON, Judith M. *The complete book of Middleware*. Boca Raton: Auerbach, c2002. ISBN 08-493-1272-8.
- [38] WARD, Brian. *How Linux works: what every superuser should know*. Second edition. San Francisco: No Starch Press, 2015. ISBN 978-1-59327-645-4.
- [39] ANDRE. Understanding Linux CPU Load - when should you be worried? *Scout* [online]. [cit. 2016-02-20]. Dostupné z: <http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages>
- [40] Understanding the Load Average on Linux and Other Unix-like Systems. *How-To Geek* [online]. [cit. 2016-02-20]. Dostupné z: <http://www.howtogeek.com/194642/understanding-the-load-average-on-linux-and-other-unix-like-systems/>
- [41] Dotaz: Load Average. *ABC Linuxu* [online]. [cit. 2016-02-20]. Dostupné z: <http://www.abclinuxu.cz/poradna/linux/show/260689>

**SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK**

ACL	Access Control List
API	Application Programming Interface
CPU	Central Processing Unit
DBAL	Database Abstraction Layer
DI	Dependency Injection
DMBS	Database Management System
ER	Entity-relationship
FAČR	Fotbalová asociace České republiky
GNU GPL	GNU General Public License
HTML	HyperText Markup Language
IO	Input/Output
JSON	JavaScript Object Notation
NoSQL	Non Relational Storage
ORM	Object-relational Mapping
PHP	PHP: Hypertext Preprocessor
SQL	Structured Query Language
SSH	Secure Shell
TCP	Transmission Control Protocol
TEMP	Temporary Folder
URL	Uniform Resource Locator
UI	User Interface
YAML	Ain't Markup Language

## SEZNAM OBRÁZKŮ

Obr. 2.1	Životní cyklus presenteru[25]	19
Obr. 2.2	Diagram tříd uživatelského účtu frameworku Nette	20
Obr. 2.3	ER diagram databáze k příkladu	24
Obr. 3.1	Příklad vzhledu datového gridu balíčku o5/grido[35]	28
Obr. 6.1	Diagram tříd procesu aktualizace struktury databáze	37
Obr. 6.2	Sekvenční diagram procesu aktualizace struktury databáze	38
Obr. 6.3	ER diagram sekcí	40
Obr. 6.4	ER diagram článku a sekce	40
Obr. 6.5	Diagram členění modulů	41
Obr. 6.6	Aktivitní diagram zpracování HTTP požadavku	42
Obr. 6.7	ER diagram příkladu databáze knih	44
Obr. 6.8	ER diagram příkladu databáze zápasu	44
Obr. 6.9	Diagram tříd struktury vrstvy Table balíčku nette/database	46
Obr. 6.10	Diagram tříd komponenty formuláře z balíčku <i>nette/forms</i>	47
Obr. 6.11	Diagram tříd vzorové struktury modelové vrstvy	51
Obr. 6.12	Diagram tříd vzorového view modelu	51
Obr. 6.13	Diagram tříd vzorové komponenty	52
Obr. 6.14	Diagram tříd komponenty Grid	52
Obr. 6.15	ER diagram modelu zabezpečení	54
Obr. 6.16	Diagram tříd implementace rozhraní IAuthorizator	54
Obr. 6.17	Diagram tříd implementace rozhraní IAuthenticator	54
Obr. 6.18	Diagram tříd komponenty převodu signálu	56
Obr. 6.19	Diagram tříd komponenty převodu presenteru	57
Obr. 6.20	Diagram tříd komponenty převodu akce	58
Obr. 6.21	Diagram tříd systému převodu presenteru	59
Obr. 7.1	Diagram generalizace presenteru	65
Obr. 7.2	Diagram tříd implementace routování	66
Obr. 7.3	Diagram tříd realizace třídy SectionRouter	67
Obr. 7.4	Sekvenční diagram znázorňující převod domény	68
Obr. 7.5	Diagram tříd realizace třídy SlugRouter	69
Obr. 7.6	Diagram tříd implementace dynamických prvků formuláře	73
Obr. 7.7	Sekvenční diagram vykreslení komponenty	76
Obr. 7.8	Diagram tříd implementace inline editace	77
Obr. 7.9	Diagram tříd realizace invalidátoru cache	78
Obr. 8.1	Graf zátěže systému	81
Obr. 8.2	Graf síťových spojení systému	82

## SEZNAM PŘÍLOH

P I.      Obsah CD

## PŘÍLOHA P I. OBSAH CD

- Technická zpráva ve formátu PDF a zdrojové texty pro  $\text{\LaTeX}$
- Zdrojové kódy realizovaných balíčků
- Skripty pro obsluhu režimu maintenance