FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Interaktivní podpora výuky v aplikaci MARAST
Student:	Rem Lohinov
Vedoucí:	Ing. Tomáš Kalvoda, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Cílem práce je rozšířit výukový webový portál MARAST o

- možnost provádět krátké ankety během výuky (učitel zobrazí anketní otázku, studenti odpovídají, následuje diskuze nad výsledky),
- možnost pokládat dotazy ze strany studentů během Q&A událostí (např. live stream).

Proveďte následující:

1. Seznamte se s frameworkem Ruby on Rails, ve kterém je MARAST napsán.
2. Proveďte analýzu požadavků potenciálních uživatelů (vyučující, studenti).
3. Navrhněte řešení jako rozšíření aplikace MARAST. Využijte již existující možnosti systému (zejména model multichoice příkladu). Dbejte na snadnost použití a přehlednost.
4. Řešení implementujte a otestujte.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 11. listopadu 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Interaktivní podpora výuky v aplikaci MARAST

Rem Lohinov

Katedra softwarového inženýrství
Vedoucí práce: Ing. Tomáš Kalvoda, Ph.D.

4. června 2020

Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Tomáši Kalvodovi, Ph.D. za cenné rady, pozitivní přístup a veškerou pomoc, kterou mi v průběhu psaní této práce poskytl. Také děkuji celé své rodině za podporu po celou dobu studia.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. června 2020 . . . . . .

Odkaz na tuto práci

Lohinov, Rem. *Interaktivní podpora výuky v aplikaci MARAST*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato bakalářská práce se zabývá přidáním technik interaktivní výuky do webového portálu MARAST, který se aktivně využívá pro podporu výuky matematických předmětů na Fakultě informačních technologií ČVUT v Praze. Obsahem práce je seznámení se s vnitřní strukturou zmíněného portálu, v něm používanými technologiemi, nezbytnou částí je také analýza požadavků potenciálních uživatelů. Cílem této práce je návrh a implementace nových funkčností zaměřených na zvětšení míry interakce mezi učiteli a studenty během výuky.

Klíčová slova MARAST, interaktivní výuka, Ruby on Rails, webová aplikace, otázky a odpovědi, kvíz, anketa

Abstract

This bachelor thesis focuses on the addition of interactive teaching techniques to the MARAST web portal, which is actively used to support the teaching of mathematical subjects at the Faculty of Information Technology CTU in Prague. The content of the work is to get acquainted with the internal structure of the portal, the technologies used in it, also an essential part is the analysis of the requirements of potential users. The goal of this work is

the design and implementation of new functionalities aimed at increasing the degree of interaction between teachers and students during teaching.

Keywords MARAST, interactive learning, Ruby on Rails, web application, questions and answers, quiz, opinion poll

Obsah

Seznam obrázků

Úvod

Výuka je extrémně důležitá, jak pro civilizaci obecně, tak pro mě osobně. Každý student by se rád dozvěděl co nejvíce za nejmenší cenu – minimum času. Naštěstí máme v současnosti hodně možností využití moderních technologií pro studium z domova (počítač, internet). Zatím jsou ale takové technologie skoro nevyužité v kontaktní výuce, což zahrnuje přednášky, cvičení, atd. Mezi výjimky patří projektory pro promítání prezentací a videí, mikrofon a reproduktory pro lepší slyšitelnost vyučujícího. Z tohoto můžeme dojít k závěru, že během kontaktní výuky jsou moderní technologie využité většinou pouze ze strany školy. Právě tuto situaci se bude snažit tato práce opravit.

Řešením mohou být techniky interaktivní výuky, jako jsou třeba online ankety, online dotazy vyučujícímu. Tyto techniky vyžadují aktivní účast studentů i učitelů a jsou, podle mého názoru, moc užitečné a pohodlné pro výuku předmětů souvisejících nejen s matematikou. Zvlášť to může být užitečné na Fakultě informačních technologií (FIT) ČVUT v Praze, kde je hodně předmětů, které jsou povinné pro všechny studenty. Současně je i studentů velmi mnoho, proto se přednášky konají ve velkých přednáškových sálech (s kapacitou do 350 lidí). Za těchto podmínek málokdo ze studentů dává aspoň nějakou okamžitou zpětnou vazbu (ve formě dotazů a komentářů), zvláště pak studenti, kteří sedí na posledních řadách posluchárny, nebo ti, kteří obecně neradi mluví před velkým počtem lidí. Kvůli tomu, že jsem během bakalářského studia byl mnohokrát svědkem tohoto smutného jevu, vybral jsem si tuto práci jako závěrečnou.

Kapitola 1

Cíl práce

Cílem praktické části práce je rozšířit webový portál MARAST (MAtematika RAdoSTně) o dvě techniky interaktivní výuky:

- Možnost pokládat dotazy ze strany studentů během Q&A událostí (nebo také přednášek, cvičení).
- Možnost provádět krátké ankety během výuky (učitel zobrazí anketní otázky, studenti odpovídají, následuje diskuze nad výsledky).

Obě zmíněné techniky jsou zaměřené na větší interakci mezi učiteli a studenty, zvláště mají zvýšit účast studentů v této komunikaci. Nezbytným cílem praktické části je také testování implementovaných funkčností, protože předpokládat bezchybnost kódu, z žádném případě, nemůžeme.

Dílčím cílem je analýza požadavků potenciálních uživatelů, což jsou učitelé a studenti. Ta poslouží k upřesnění konkrétních požadavků na tato rozšíření, dá přehled toho, co by uživatelé nejvíce ocenili.

Cílem textové části této práce je seznámit se s frameworkem[1] Ruby on Rails, pomocí kterého je MARAST implementován, s vnitřní strukturou toho webového portálu samotného, moderními metodami vývoje webových aplikací, které následně budou použity při implementaci. Také se v této části má zjistit technologie, která se využívá pro testování dotyčného webového portálu. Tato technologie se bude využívat pro testování za účelem konzistence.

[1] Framework je knihovna funkcí pro usnadnění vývoje aplikace [1].

Kapitola 2

Analýza

Tato část bude věnovaná analýze v širším slova smyslu. Bude rozdělená na pět částí:

1. Stručné představení jazyku Ruby.
2. Představení frameworku Ruby on Rails, pomocí kterého je MARAST implementován. Zde budou uvedené důležité principy a metody vývoje aplikací ve zmíněném frameworku.
3. Představení webového portálu MARAST, jeho vnitřní struktury, použitých rozšířeních frameworku.
4. Analýza požadavků potenciálních uživatelů naplánovaných funkčností.
5. Finální požadavky na rozšíření.

2.1 *Ruby*

*Ruby*je interpretovaný skriptovací programovací jazyk, který vznikl v roce 1995. Jeho vývoj začal v roce 1993, a v této době jeho jediným tvůrcem byl japonský programátor Yukihiro Matsumoto také známý pod přezdívkou „Matz“ [2]. Nejdříve je vhodné si vysvětlit význam termínů využitých v uvedené definici.

Interpretovaný znamená to, že pro spouštění programu je nezbytný jeho zdrojový kód a zvláštní program zvaný interpret, který zdrojový kód provádí [3]. Mezi výhody tohoto přístupu lze zařadit obvykle snadnou přenositelnost programu mezi platformami (stačí použít správného interpreta) a snazší hledání a odstraňování chyb [3]. Tento přístup má své nevýhody, některé z nich, dokonce, souvisejí v uvedenými výhodami. To platí kupříkladu pro nutnost přítomnosti interpreta konkrétního jazyka pro konkrétní platformu, další nevýhodou je pomalejší běh programu (v porovnání s kompilovanými jazyky, které používají druhý populární způsob spouštění programů) [3].

Skriptovací znamená to (i když přesná definice neexistuje), že jazyk je navržen především k automatizaci úloh, k manipulaci s prostředky stávajícího systému, případně k jejich uzpůsobování potřebám zákazníka nebo uživatele [4]. Alternativně úkoly řešené skriptovacím jazykem by mohly být vyřešené člověkem jeden za druhým [5].

2.1.1 Historie a popis

Autor jazyka *Ruby*, jako zastánce objektově orientovaného programování, hledal v první polovině 90. let skriptovací jazyk, který by mu vyhovoval. Jelikož žádný takový nenašel, došlo k vytvoření Ruby [2]. Jak říká sám autor, „I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That's why I decided to design my own language“ [6]. Tato snaha ho přivedla k tomu, že v dnešní době, skoro po 30 letech vývoje, je Ruby moderní, dynamický, objektově orientovaný jazyk se zaměřením na jednoduchost a produktivitu. Nabízí elegantní syntaxi, která je přirozená pro psaní a snadno čitelná [7]. Pojem „dynamický“ v tomto případě znázorňuje kupříkladu to, že *Ruby* používá dynamické typování proměnných, což je charakteristický rys většiny dynamických jazyků [8]. Původní snaha autora o objektově orientovaný jazyk byla zcela naplněna, jelikož všechno v Ruby je objekt. Všechno znamená, že dokonce jednoduché číselné literály a hodnoty true, false, nil (nil je speciální hodnota, která označuje nepřítomnost hodnoty) jsou objekty [9].

V dnešní době je *Ruby* jedním z nejpopulárnějších jazyků mezi technologickými startupy[2]. Hodně celosvětově známých firem, jako jsou *Airbnb, Twitch, GitHub, Twitter* používají ve svých projektech Ruby [11].

2.2 *Ruby on Rails*

Ruby on Rails je framework pro vývoj webových aplikací napojených na databázi, používající architekturu *model-view-controller* (tato architektura bude popsána v kapitole 2.2.1). Framework vytvořil dánský programátor David Heinemeier Hansson, je implementován v jazyce *Ruby*, a na tomto jazyce je všechno v *Ruby on Rails* založeno [12].

Poprvé byl tento framework vydán v roce 2005 [12] a od té doby se rychle stal jedním z nejpopulárnějších nástrojů pro vytváření webových aplikací. *Ruby on Rails* stejně jako Ruby používá hodně celosvětově známých firem, mezi které patří *Airbnb, SoundCloud, Disney, Hulu, GitHub, Shopify* [13].

Ruby on Rails má pro programátora velkou výhodu v tom, že na rozdíl od některých jiných vývojových nástrojů (kupříkladu *JavaScript/Node.js*) s časem zachovává velmi podstatnou část používaných technologií. I když, na-

[2]Startup je společnost designovaná pro rychlý růst [10]

štěstí, toto neznamená, že tento framework nikdo nevyvíjí, protože nové verze vycházejí pravidelně [14].

Klasická a správná aplikace v *Ruby on Rails*, kromě MVC, používá i jiné principy [14], mezi které patří:

1. DRY, neboli *don't repeat yourself*. Tento princip spočívá ve snížení duplicitních částí kódu v rámci aplikace.

2. CoC, neboli *convention over configuration*. Tento princip spočívá ve snížení počtu rozhodnutí, která má dělat vývojář [15]. To je docíleno velkým počtem doporučení a hotových řešení, která stačí jenom použít na řešený problém. Kupříkladu to může být standardní pro všechny projekty, strukturu složek a rozmístění souborů.

2.2.1 *Model-View-Controller*

Model-View-Controller je softwarová architektura často používaná při vývoji webových aplikací jakožto aplikací poskytujících uživatelské rozhraní přístupné přes internet [16]. Základní myšlenkou tohoto konceptu je rozdělení datového modelu, uživatelského rozhraní a řídicí logiky aplikace do tří komponent [16], každá z nich má svou konkrétní roli:

- ***View* (pohled)** – komponenta zodpovědná za zobrazení dat, která dostala od řadiče, ve formě vhodné pro konečného uživatele (třeba jako webová stránka, nebo tabulka).

- ***Controller* (řadič)** – představuje mezivrstvu, která je zodpovědná za zpracování (může tady být i jednoduchá validace[3]) vstupu od uživatelů a přeposlání zpracovaného vstupu do modelu. V některých případech, když pro zobrazení výsledného pohledu není nutná komunikace s databází, může rovnou zobrazit požadovaný pohled [13].

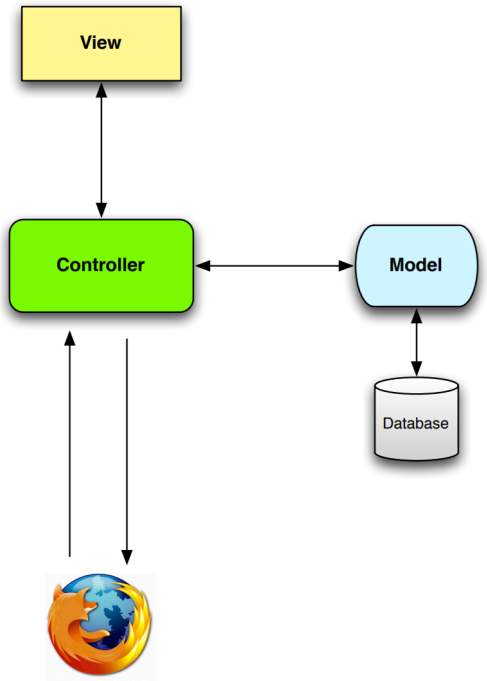
- ***Model* (model)** – hlavní rolí této komponenty je komunikace s databází. Konkrétní vstupy, které zpracovává, dostává od řadiče, také tuto vstupy validuje. Každý model může mít víc pohledů, které zobrazují data (poskytnuté tímto modelem v různých formátech).

Výsledkem tohoto rozdělení je následující postup zpracování požadavku od uživatele, ten je klasický pro *Ruby on Rails* aplikace (je uveden v knize [13]):

1. Odeslání požadavku prohlížečem. Tohoto může být docíleno například otevřením webové stránky.

2. Po přijetí serverem je tento požadavek předán řadiči.

[3] Validace je kontrolou, ověřením správnosti vstupních dat.



Obrázek 2.1: MVC architektura (převzato z [13])

3. Řadič, když je potřeba, komunikuje s modelem.

4. Model provádí s databází požadovanou operaci (kupříkladu, čte data).

5. Použije se konkrétní pohled pro vygenerování webové stránky (ta se doplní pomocí dat získaných modelem, která pohled dostal od řadiče).

6. Řadič předává prohlížeči výslednou stránku.

Uvedený postup je znázorněn na obrázku 2.1.

Editace každé z těchto (model, řadič, pohled) komponent má ovlivňovat ostatní jenom minimálně. Výsledkem tohoto rozdělení, při správné realizaci, je lépe udržovatelný a testovatelný systém méně náchylný na chyby. Méně náchylný na chyby a lépe udržovatelný je proto, že za použití této architektury dochází ke značné míře znovupoužitelnosti kódu. [17]. Lépe testovatelný je proto, že každá část může být velmi podrobně testována zvlášť [17].

Další výhodou použití MVC je možnost paralelního vývoje, což znamená, že více vývojářů může pohodlně vyvíjet MVC aplikaci zároveň, což zřejmě podporuje rozdělená architektura [16].

2.2.2 *MVC v Ruby on Rails*

Ruby on Rails pro jednu tabulku v databázi používá jeden model, jeden řadič a více pohledů [1]. Důvodem, proč je pohledů více je to, že se používají, kupříkladu, pro zobrazení, vytvářecí formulář, editační formulář atd.

Také framework *Ruby on Rails* používá konvenci pojmenování jednotlivých částí architektury MVC spočívající v pojmenování modelů v jednotném čísle a řadičů v čísle množném [13]. Kupříkladu by se model jmenoval *State* (stát), ale odpovídající řadič *StatesController* (řadič států).

Jelikož tabulek v databázi je obvykle vícero, je vícero i řadičů, které jim odpovídají. Proto je nutné upřesnit postup vyřízení požadavků uvedených v sekci 2.2.1 protože tam jsme pracovali s jediným řadičem). Zde vstoupí „do hry“ tzv. *Router* (směrovač).

***Router* (směrovač)**

Tato součástka *Ruby on Rails* projektů je zodpovědná za rozhodnutí o tom, jaký požadavek bude zpracovávat jaký řadič a jakou metodou (to bude popsáno v sekci 3.3). Konkrétně právě směrovač představuje „hlavní bránu“, přes kterou procházejí všechny požadavky.

2.3 MARAST

Výukový webový portál MARAST byl vytvořen během zimního semestru akademického roku 2012/2013 [18]. Jeho původním cílem byla podpora výuky matematiky na FIT ČVUT, především se měl použít pro výuku matematických předmětů určených pro první ročník bakalářského studia. Mezi těmi předměty patřily: „Základy matematické analýzy“, „Přípravný kurz matematiky“, „Lineární algebra“ [19]. S postupem času se seznam podporovaných předmětů rozšiřoval a k roku 2017 zahrnoval 7 předmětů vyučovaných na dotyčné fakultě [18]. K roku 2020 se tento seznam rozšířil ještě, minimálně, o předmět „Automaty a gramatiky“, což autor využil během studia.

Hlavní součástí MARASTu vždy byla a je sbírka příkladů a dalších studijních materiálů, které studenti a učitelé mohou využít pro pohodlnější výuku.

Na začátku své historie MARAST měl velmi omezenou část dnes podporovaných funkčností. Pro studenty v té době bylo jedinou možností využití systému při řešení přidaných příkladů, provádění základního vyhledávání v nich. Učitelé mohli zmíněnou sbírku příkladů také využít pro sestavení písemných testů [1].

Časem se ale MARASTu přidávaly i jiné funkčnosti, některé z nich implementoval vývojářský tým, některé studenti FIT ČVUT v rámci svých bakalářských a diplomových prací [18].

V současné době MARAST, kromě již popsaných funkčností, podporuje, kupříkladu:

- **Kvízy** představují specifické skupiny příkladů, které se týkají konkrétního předmětu. Některé z kvízů jsou povinné a student je má splnit, aby získal nárok na zápočet z dotyčného předmětu. Hodně kvízů je časově omezených (studenti k nim mají přístup jenom omezenou dobu) a každý kvíz má nastavenou reakci na špatnou odpověď ze strany studenta (to může být třeba zablokování možnosti zobrazení další otázky na nějakou dobu).

- **Lekce** představují množinu materiálů souvisejících s nějakým tématem z nějakého předmětu. Sem patří vyučovací textové články, videa, sbírky příkladů pro procvičení.

- **Komentáře**, které se dají přidat například k příkladům v cvičebnici. Ty se rozdělují na tři typy: obecný komentář, otázka, odpověď. Podle tohoto rozdělení se zobrazuje každý jinak, uživatelé mají s nimi různé možnosti interakcí (nedá se hlasovat o obecný komentář a otázku, jenom o odpověď). Zmíněné hlasování má podobu „líbí se mi“, „nelíbí se mi“. Tyto hlasy se potom využívají pro určení pořadí ze seznamu komentářů.

2.3.1 Struktura MARASTu

Jak už bylo zmíněno, MARAST je založen na frameworku *Ruby on Rails* a používá architekturu MVC.

Celkem ke dni 27. března 2020 má produkční verze MARASTu 28 modelů. Uvádět popis jich všech by nemělo smysl, protože toto rozšíření bude pracovat jenom s jejich malou částí. Modely a řadiče, se kterými se bude pracovat, jsou popsané v kapitole 3.

Dále jsou popsaná důležitá rozšíření frameworku *Ruby on Rails* používaná MARASTem.

Přístupová práva

Jelikož se portál MARAST využívá pro hodnocení studentů, všechny jeho části mají být chráněny, aby byly přístupné jen oprávněným uživatelům. Pro nastavení přístupových práv MARAST používá rozšíření *Pundit*[4]. Toto rozšíření funguje na principu, že pro model (přístupové právo, ke kterému chceme nastavit) vytvoříme třídu, ve které budou popsaná přístupová práva k jednotlivým metodám (akcím) v řadiči. Podle konvencí se tato třída s popisem přístupových práv nazývá „Název modelu“+„Policy“.

Kupříkladu, když implementujeme model *State* (stát) a v řadiči máme metodu pro zobrazení již existujícího státu:

```
def show
  @state = State.find(params[:id])
```

[4] `https://github.com/varvet/pundit`

```
    authorize @state
  end
```

Potom ve třídě *StatePolicy* ověříme, zda aktuální uživatel má přístup k tomuto záznamu:

```
  def show?
    user.admin?
  end
```

V tomto případě uživatel má přístup jenom tehdy, když je administrátorem (toto je uvedeno jenom za účelem snadnosti pochopení, v praxi by nejspíše byla podmínka komplikovanější).

Formy studia

Pro nastavení forem studia uživatelů v rámci předmětů používá MARAST speciální tabulku Role (i když to není jediný smysl této tabulky). Záznamy v ní jsou vztažené ke konkrétnímu uživateli, semestru a předmětu. Existuji 4 různé formy studia: „Neurčená“, „Prezenční“, „Kombinovaná“, „Anglická“. Učitele vždy mají „neurčenou“ formu studia. Toto je důležité, protože podle požadavků na rozšíření máme s formami studia pracovat (2.5).

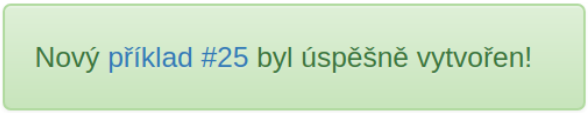
Testování

Pro provádění automatických testů (ty budou popsané v kapitole 5) MARAST využívá rozšíření *Ruby on Rails*, které se nazývá *RSpec*[5]. Pomocí tohoto rozšíření se dají testovat všechny tři komponenty architektury MVC (model, pohled, řadič). Základem testování ale je vytvoření testovacích instancí tříd, se kterými potom provádíme manipulace.

Testovací objekty

Pro vytvoření testovacích objektů MARAST využívá jiné rozšíření *Ruby on Rails*, které se nazývá *Factory Bot*[6]. Používá to místo standardně poskytovaného *Ruby on Rails* mechanizmu, zvaného *fixture*. Ten představuje způsob testování pomocí testů, které jsou pro všechny společné, databáze se definuje a naplňuje ještě před spuštěním prvního testu. Factory bot má jiný přístup, který dovoluje vytvoření nutných dat pro test před spuštěním každého testu zvlášť, což odstraní chyby související se závislostmi mezi testy [20].

[5] `https://github.com/rspec/rspec-rails`
[6] `https://github.com/thoughtbot/factory_bot`



Obrázek 2.2: Příklad hlášky v MARASTu

CSS styly

Pro nastavení stylů, pohledů a hlášek používá MARAST další CSS framework *Bootstrap*[7]. Pod pojmem hláška se myslí zpráva, která dává uživateli vědět, zda nějaká akce byla úspěšně provedena, příklad je na obrázku 2.2. Pomocí tohoto rozšíření se také řeší responzivní design (to, jak webové stránky budou vypadat na různých obrazovkách). Smyslem tohoto rozšíření je velký počet přípravných stylů pro různé elementy uživatelského rozhraní. Sem patří kupříkladu styly pro tlačítka, navigační lišty (ty se často uvádějí nahoře stránky, obsahují odkazy na důležité komponenty webové aplikace), formuláře atd. Programátor potom musí provést jenom relativně drobné změny pohledu stránek, aby odpovídaly jeho představě.

2.4 Analýza požadavků potenciálních uživatelů

Tato kapitola je zásadní pro celou bakalářskou práci, protože obsahuje analýzu požadavků vyučujících a studentů. Zásadní je z toho důvodu, že právě z analýzy požadavků vyučujících vyplynula možnost o implementaci tohoto rozšíření v rámci bakalářské práce. Bude rozdělena na dvě podkapitoly podle zmíněných skupin respondentů.

2.4.1 Učitelé

Pro vyučující byl využit speciální chat s omezeným přístupem na platformě *GitLab*[8], kde učitelé diskutovali o tom, co přesně aplikace má umět. Ukázalo se, že toto řešení je velice efektivní, protože i přes omezený počet účastníků konverzace (celkem 6) bylo získáno mnoho nápadů. Z tohoto chatu jsem extrahoval konkrétní požadavky na rozšíření, odkaz, který byl potom přidán do původního chatu, kde učitelé finálně mohli zkontrolovat, co bylo vypsáno. Původně v tomto chatu byl uveden velký počet požadavků na rozšíření, ale po několika konzultacích s vedoucím práce byl vybrán seznam nejzásadnějších, které tato práce bude následně implementovat. Ty mají vytvořit dobrý základ, který se potom dá jednoduše rozšiřovat.

Mimo jiné existoval v tomto chatu požadavek na vytvoření bodované verze ankety. Po konzultaci s vedoucím práce a jeho účasti ve zmíněném chatu bylo

[7] `https://getbootstrap.com`
[8] https://about.gitlab.com/

ale rozhodnuto, že pro řešení tohoto požadavku stačí využít jen částečně upravený model Kvíz (popsaný v kapitole 2.3), což méně souvisí s obsahem ostatní části této práce, proto tento požadavek práce neřeší.

2.4.2 Studenti

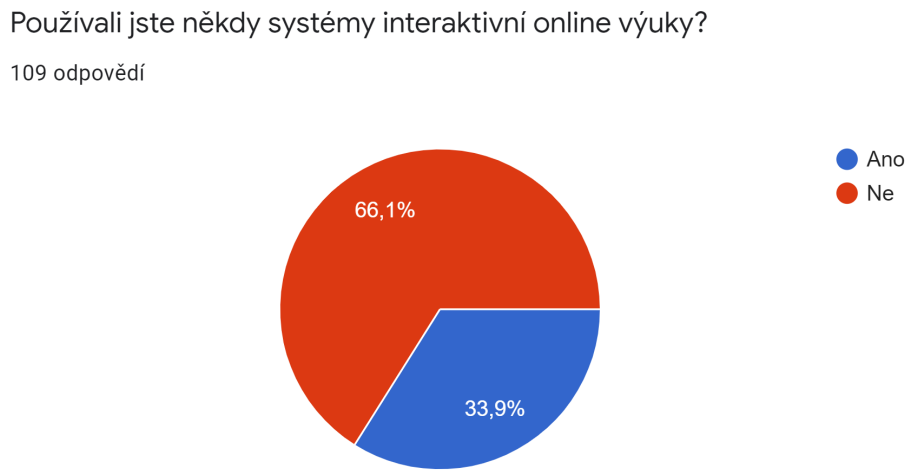
Obrázek 2.3: Odpovědi na první otázku v dotazníku

Pro studenty byl vytvořen online dotazník na platformě *Google Forms*. Ten byl publikován ve studentských skupinách na Facebooku a rozeslán do chatů studentů. Pro zvýšení počtu odpovědí, bylo rozhodnuto tento dotazník vytvořit anonymním. Ke dni 3. května ten měl 109 dokončených odpovědí, což svědčí o zájmu ze strany studentů. Ihned první otázka se studentů ptala, odkud se o tomto dotazníku dozvěděli. Jak je vidět z obrázku 2.3, skoro všichni se o něm dozvěděli ze studentských skupin na Facebooku. Jednou z příčin tohoto výsledku je aktivní účast studentů v těchto skupinách. Druhou je to, že jsem odkaz na tento dotazník posílal nejvíce právě tam.

Systémy interaktivní online výuky

Druhá otázka v dotazníku zněla **„Používali jste někdy systémy interaktivní online výuky?“** Na obrázku 2.4 je znázorněn diagram odpovědí na tuto otázku. Z odpovědí plyne, že dvě třetiny respondentů nikdy nepoužívali podobné systémy, což dává najevo to, že zatím na FIT ČVUT takové systémy často používané nejsou. Také statistika ukazuje důležitost této bakalářské práce (jejíž cílem je přidání takového systému).

Dále následovaly otázky typu „Co jste používali?“, „Co se Vám líbilo?“, „Co se Vám nelíbilo?“. Z odpovědí na tyto otázky lze pochopit, že se nejvíce studentům líbil snadný přístup, uživatelská přívětivost, přehlednost výsledků a možnost klást anonymní otázky. Mezi nedostatky použitých systémů zařadili respondenti to, že tyto systémy neměly pro ně důležité funkčnosti (nebo měly, ale byly zpoplatněny), zvláště co se výuky matematiky týče.



Obrázek 2.4: Odpovědi na druhou otázku v dotazníku

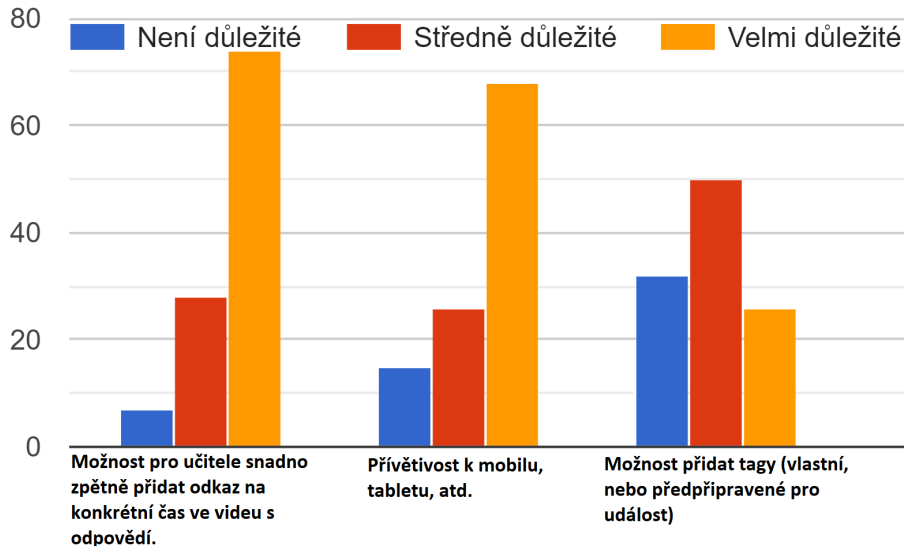
Dotazy během Q&A událostí

Další otázka se vztahovala k dotazům během Q&A událostí, zněla **„Co byste v systému nejvíce ocenili?“** a obsahovala 11 podotázek, kde byly popsány jednotlivé požadavky, které byly extrahované ze sekce 2.4.1 (požadavků učitelů). Studenti měli pro každou podotázku 3 varianty odpovědí: „Není důležité“, „Středně důležité“, „Velmi důležité“. Zde budou popsané podotázky s nejradikálnějšími histogramy[9], protože ty si zaslouží největší pozornost.

Podotázkou s největším počtem odpovědí „Velmi důležité“ byla „Možnost pro učitelé snadno zpětně přidat odkaz na konkrétní čas ve videu s odpovědí“, podobný histogram měla otázka „Přívětivost k mobilu, tabletu, atp“. Tyto histogramy jsou uvedené na obrázku 2.5. Výsledek první otázky vypovídá o zájmu studentů o použití naplánovaných dotazů během nahrávaných přednášek (nebo konzultací, proseminářů atd.), což se na FIT ČVUT využívá. Studenti mají například k dispozici nahrávky přednášek z předmětů „Základy matematické analýzy“, „Lineární algebra“. Výsledek druhé zmíněné otázky vypovídá o tom, že studenti chtějí používat naplánované rozšíření pomocí různých zařízení, z čehož vyplývá důležitost tohoto požadavku.

Podotázkou s největším počtem odpovědí „Není důležité“ a nejmenším počtem „Velmi důležité“ byla podotázka „Možnost přidat tagy (vlastní, nebo předpřipravené pro událost)“, histogram je také uveden na obrázku 2.5. Tento výsledek byl neočekávaným, protože vyučující považovali tento požadavek za stejně perspektivní jako ostatní (žádné negativní názory o tom nebyly v jejich chatu vypsané). Kvůli této zpětné vazbě ze strany studentů tento funkční požadavek byl vyloučen ze seznamu těch, které se mají implementovat v první řadě (v rámci této bakalářské práce).

[9] „Histogram je grafické znázornění distribuce dat pomocí sloupcového grafu“ [21]



Obrázek 2.5: Radikální odpovědi studentů

Ankety během výuky

Poslední otázka v dotazníku byla ohledně anket a zněla **„Co si myslíte, že je důležité právě pro tuto sekci? Jaké možnosti byste požadovali? Čeho bychom se měli vyhnout?“**.

Tuto otázku bylo rozhodnuto udělat otevřenou, protože představa o tom, jak to má fungovat, byla u učitelů většinou definitivní a ta byla popsáná v záhlaví dotazníku takto: „Možnost provádět krátké ankety během výuky (učitel zobrazí anketní otázku, studenti odpovídají, následuje diskuze nad výsledky)“. Stejně ale bylo důležitým dostat poznámky od studentů.

V odpovědích se opakoval požadavek na přehlednost a jednoduchost použití. Studenti zvýrazňovali důležitost anonymních odpovědí, také prosili o grafické znázornění četnosti jednotlivých variant řešení. Také byl uveden požadavek na to, aby se studenti mohli dívat na své minulé odpovědi.

2.5 Finální požadavky

V této kapitole budou popsány finální požadavky, ty vyplývají ze zadání bakalářské práce a jsou upřesněné pomocí analýzy požadavků vyučujících a studentů. Kapitola bude rozdělená na tři části: první bude věnovaná obecným požadavkům na rozšíření a událostem, druhá požadavkům na dotazy během událostí, třetí požadavkům na ankety.

2.5.1 Obecné požadavky na rozšíření a události

1. Podpora sázecího systému *LaTeX* pro matematické vzorce a značkovacího jazyku *Markdown*.
2. Možnost k tělu události přiložit snímek obrazovky/fotografii.
3. Přívětivost k mobilu, tabletu, atd.
4. Konzistence s ostatní části systému, využití jeho možností.
5. Podpora anglického jazyka, jakož možného jazyka uživatelského rozhraní.
6. Archiv událostí přístupných uživateli.
7. Rozdělení archivu na semestry.
8. Rozdělení událostí podle forem studia (prezenční, kombinovaná atd.).

2.5.2 Dotazy během Q&A událostí

1. Možnost pokládání otázek od studentů.
2. Možnost studentů „lajkovat“ (palec nahoru, dolu) otázky.
3. Možnost automatického řazení otázek podle hlasů studentů.
4. Možnost pro učitele manuálně řadit otázky lokálně (na svém počítači).
5. Možnost zvýraznění právě zodpovídané otázky.
6. Rozdělení otázek na aktivní a neaktivní.
7. Nekontaktní varianta - možnost textové odpovědi na otázky.
8. Možnost pro učitelé snadno zpětně přidat odkaz na konkrétní čas ve videu s odpovědí.
9. Podpora anonymních dotazů.

2.5.3 Ankety během výuky

1. Možnost studentů odpovídat na otázky.
2. Podpora multichoice (jedna správně, více správně), textových otázek.
3. Rozšíření multichoice o větší počet možností (10).
4. Různé varianty zobrazení výsledku.

5. Povolení studentům odpovídat na otázky jen během definovaného intervalu času.
6. Možnost prohlížení odpovědí konkrétního studenta (jen v případě neanonymní události).
7. Anonymní varianta (kde nebude vidět, jak odpovídal konkrétní student).

Kapitola 3

Návrh

Jak už bylo uvedeno v kapitole 2.3 (věnované analýze), MARAST využívá softwarovou architekturu MVC. Za účelem konsistence a návaznosti na už implementované možnosti systému, bylo rozhodnuto se této architektury držet i v tomto rozšíření.

Tato kapitola bude věnovaná návrhu všech třech komponent architektury MVC. Na začátku bude uveden návrh databázového modelu. Díky konvencím používaným v *Ruby on Rails* (popsáno v kapitole 2.2.2), ten se bezprostředně použije jako návrh modelů v rámci MVC. Potom bude uveden návrh pohledů nutných pro implementaci funkčních požadavků na naplánované rozšíření. Na konci bude uveden návrh řadičů, které mají svázat modely a pohledy. V této kapitole budou uvedené jenom modely, pohledy a řadiče, které před začátkem této práce v MARASTu nebyly, tj. zde nebudeme uvádět to, jak je naplánováno měnit, či používat již existující části systému, tento popis necháme na implementační fázi.

Zásadním detailem který vyplynul z analýzy požadavků potenciálních uživatelů bylo přidání také anonymní verze dotazů a anket. Tento požadavek považuji za velice úspěšný, protože u Q&A událostí to má přispět k většímu počtu dotazů od studentů. U anket k tomu, že studenti nebudou mít strach odpovědět špatně, což má zmenšit počet podvodů (studenti nebudou mít k tomu motivaci). V této kapitole máme navrhnout možné řešení tohoto požadavku.

3.1 Návrh databázového modelu

Návrh databáze je základem celého řešení. Jeho důležitost je těžko přecenit, protože kolem tohoto se potom bude stavět celá aplikace, nebo v našem případě, celé rozšíření. Pokud tento návrh proběhne úspěšně, může značně usnadnit další vývoj. V opačném případě se může vývoj významně zkomplikovat, nebo dokonce může dojít k tomu, že správným řešením bude celý systém navrhnout znovu, což může znamenat velkou ztrátu času [1].

Před návrhem nových tabulek provedl autor detailní analýzu aktuální verze MARASTu (částečně popsané v kapitole 2.3). Jelikož je MARAST v současnosti už velkým projektem, který se rozvíjel od roku 2012, má implementováno hodně funkčností, kterých jsem se snažil co nejvíce ve svém návrhu využít. Proto jsem pro implementaci požadovaných funkčností navrhl vytvoření jenom 3 nových tabulek:

1. *Event* (událost), popsáno v 3.1.1
2. *PollQuestion* (otázka v anketě), popsáno v 3.1.2
3. *PollAnswer* (odpověď na otázku v anketě), popsáno v 3.1.3

Ty budou popsané dále v této kapitole.

Jako otázky v událostech typu Q&A jsem se rozhodl využít už existující model **Comment (komentář)**. Po analýze a konzultaci s vedoucím práce jsem si uvědomil, že ten lze relativně jednoduše upravit, aby odpovídal požadavkům na takové otázky. Tyto úpravy budou uvedeny v kapitole 4.

Důležitým detailem, který je potřeba uvést už zde, je to, že v zadání bakalářské práce je napsáno **„Využijte již existující možnosti systému (zejména model multichoice příkladu)“**. I když jsem se snažil během návrhu maximálně využívat již existujících možností systému, rozhodl jsem se nevyužívat konkrétně model multichoice příkladu. Důvody, proč jsem se takto rozhodl, a také vlastní návrh řešení, budou popsané v kapitole 3.1.2.

3.1.1 *Event*

Tato tabulka představuje obecnou událost, kam patří Q&A události (během kterých studenti mohou klást dotazy vyučujícímu), a také události, během kterých studenti odpovídají na otázky v anketách. Tyto dva typy funkčností jsem se rozhodl sjednotit do jednoho modelu (tabulky), protože po hlubší analýze požadavků a konzultací s vedoucím práce, jsem pochopil, že mají docela hodně společných atributů a odlišují se jenom v několika. Dále bude uveden seznam a popis atributů této tabulky.

Kind (typ)

Je typu **enum** a používá se pro **typ** události. Tento atribut požívají oba typy událostí, je povinným. Může nabývat hodnot „q_and_a“ a „poll“ (Anketa). V závislosti na tomto atributu budeme s událostí pracovat vhodným způsobem.

Title

Je typu **string** a používá se pro **název** události. Tento atribut požívají oba typy událostí a je povinným.

Abstract

Je typu **string** a používá se pro **abstrakt** události. Tento atribut požívají oba typy událostí, je povinným. Přidán byl kvůli tomu, že název většinou nestačí k vysvětlení toho, čemu se tato událost bude věnovat. Udělení tohoto atributu povinným byl požadavkem vedoucího práce, jakož garanta předmětu „Základy matematické analýzy“, vyučovaného v MARASTu. Toto má donutit učitele předmětů, aby důkladněji vysvětlovali obsah události.

Body

Je typu **string** a používá se pro **tělo** události. Tento atribut požívají jenom události typu Q&A, proto není povinný. Zde může být uveden libovolný obsah související s událostí, například podrobný časový plán přednášky nebo alternativa slajdům promítaných na projektoru. Obecně je zde ponechán prostor pro fantazii učitelů. Všechny podporované možnosti dostupné učitelům pro tvorbu tohoto doprovodného materiálu budou popsány v kapitole 4.

Anonymous_responses

Je typu **boolean** a požívají ho oba typy událostí, je povinným. U různých typů událostí se využívá jinak. Události typu Q&A ho používají pro nastavení toho, zda dotazy v odpovídající události budou anonymní nebo ne. Události typu Poll ho používají pro nastavení anonymity odpovědí na otázky patřící k této události.

Mod_of_inquiries

Je typu **enum** a požívá se událostmi typu Q&A pro nastavení pořadí zobrazení komentářů, povinným není. Pro splnění zadání práce bude tento atribut moci obsahovat dvě „klasické“ varianty ražení:

- podle času vytvoření komentářů
- podle hlasů (poměr „palců nahoru a dolů“)

Start, end

Tyto dva atributy jsou typu **datetime** a používají se pro nastavení času začátku a konce událostí obou typů. Při implementaci budeme moci tyto atributy využít pro rozdělení všech událostí přístupných studentům podle toho, zda jsou už minulé, živé (aktivní) nebo budoucí. Toto se pak využije pro definování povolených a nepovolených akcí s konkrétní událostí v konkrétní okamžik času. Kupříkladu podle požadavků uvedených v kapitole 2.5.3 mohou studenti odpovídat na otázky v anketě pouze v uvedený časový interval. Také podle požadavku vedoucího práce student nemůže otevřít stránku budoucí události (i když ji vidí v seznamu).

Semester_id, course_id

Tyto dva atributy jsou typu **integer**, protože vlastně představují cizí klíče[10] vztažené k tabulkám Semester (semestr) a Course (předmět)se využijí v obou typech událostí, tyto atributy ale nejsou povinné. Využití těchto atributů dovolí omezit možnost přístupu studentů jenom k událostem předmětů, které mají zapsané, a to s přihlédnutím semestru, ve kterém jsou zapsané. Právě kvůli tomuto naplánovanému omezení nejsou tyto atributy povinné (kupříkladu, aby bylo možné vytvořit události přístupné studentům nezávisle na semestru, ve kterém ten předmět mají zapsán). Podrobnější popis je v kapitole 4.

Form_of_study

Je typu **enum**, používají ho události obou typů, je povinným atributem. Možnými jsou klasické 4 formy studia používané v MARASTu: „Neurčená“, „Prezenční“, „Kombinovaná“, „Anglická“. Výchozí hodnotou je „neurčená“ forma studia.

User_id

Tento atribut představuje autora události, je typu **integer**, protože je cizím klíčem vztaženým k tabulce User (uživatel). Tento atribut je povinným pro oba typy událostí a během implementací ho budeme moci využít pro nastavení práv na editaci, mazaní událostí atd.

3.1.2 *PollQuestion*

Tato tabulka představuje otázky v anketě (využívá se v událostech typu Anketa). Rozhodl jsem si ji vytvořit navzdory tomu, že v zadání této práce je uveden požadavek na využití modelu multichoice příkladu, který je součástí modelu *Problem* (příklad). Po provedení analýzy a konzultaci s vedoucím práce, jsem si pochopil, že v aktuální verzi MARASTu model *Problem* už je velmi komplikovaný, zdrojové kódy modelu a jeho řadičů jsou jedny z nejrozsáhlejších v celém systému, počet pohledů je také jedním z největších. To ale vůbec neznamená, že budeme moci požadované funkčnosti jednoduše implementovat, protože by se nedalo využít skoro nic bez speciálních úprav.

Obecně, „příklad“ v cvičebnici a „otázka v anketě“ jsou významově vlastně hodně jiné věci. Příklad člověk může řešit i celý den, u otázky v anketě se to nepředpokládá.

A jelikož *Problem* je centrálním modelem celého MARASTu, znamenalo by to, že bude potřeba důkladně otestovat, že nové funkčnosti nemění staré tam, kde nemají. Navíc by to komplikovalo i tak těžce pochopitelný model.

[10] Cizí klíč definuje vztah mezi dvěma tabulkami, který znamená, že záznam v jedné tabulce je nějakým způsobem závislý na záznamu v druhé tabulce

Proto bychom mohli přidat požadované funkčnosti mnohem jednodušeji, pomocí nového modelu, který příliš náročným nebude. Dále uvedu popis atributu tabulky odpovídající tomuto novému modelu.

Kind

Je datového typu **enum** a je povinným, představuje typ otázky. Podle požadavků na rozšíření, uvedených v 2.5.3, chceme umožnit 3 typy otázek:

- ***Singlechoice*** obsahuje několik možných odpovědí. Student si z nich má vybrat právě jednu.
- ***Multichoice*** obsahuje několik možných odpovědí. Student si z nich může vybrat libovolný počet.
- **Textová** obsahuje otázku a místo pro odpověď.

Proto tento atribut může nabývat hodnot „*single_choice*“, „*multi_choice*“ a „*text_field*“.

Event_id

Je typu **integer**, je cizím klíčem vztaženým k tabulce Event. Představuje událost, které náleží tato otázka. Jelikož otázky nevztažené k událostem plánované nejsou, tento atribut je povinným.

Body

Je typu **string** a představuje zadání úkolu, například „2*2=“. Tento atribut používají všechny typy otázek, a jelikož otázka bez zadání žádný smysl nemá, je tento atribut povinným.

Option0, option1, . . . , option9

Tyto atributy jsou typu **string**, a představují jednotlivé možnosti odpovědí. Povinnými být nemohou, protože mají nějaké použití jenom pro otázky typu „single_choice“ a „multi_choice“.

Diskutabilní otázkou ale je, jestli tyto možnosti zapisovat takto zvlášť do různých atributů, nebo je sjednotit do jednoho ve formátu JSON[11], který lze následně zpracovávat a rozkládat na jednotlivé možnosti odpovědí. Ve svém návrhu jsem se rozhodl držet klasického databázového přístupu (jednotlivých atributů), protože neexistuje žádný důvod, kvůli kterému bych od toho přístupu odcházel. Takovým důvodem by mohl být velký maximální počet možností odpovědí. Ale v konverzaci učitelů zmíněné v kapitole 2.4.1 (analýza

[11] JSON je datový formát, který umožňuje ukládání a přenos libovolné datové struktury, kterou dokážeme organizovat v polích nebo agregovat v objektech [22].

požadavků)bylo požadováno maximálně 10 možností, takže tento důvod relevantním není. Dalším důvodem by mohlo být to, že chceme do tohoto atributu ukládat data s různorodou strukturou, což v našem případě také pravda není.

Number_of_options

Je typu **integer** a představuje počet možností u otázek typu „single_choice“ a „multi_choice“. Povinným tento atribut není, protože se nevyužije pro otázky s textovým políčkem.

Result_json

Atribut je typu **string** ten navrhuji použít pro ukládání agregovaných odpovědí na dotyčnou otázku, povinným není. Typu string je proto, abychom v něm mohli ukládat data ve formátu JSON, ten už se sem velice hodí. Pracovat s ním navrhuji tak, že pro otázky typů „single_choice“ a „multi_choice“ budeme ukládat pole 10 čísel, kde i-té číslo bude znamenat počet odpovědí, ve kterých i-tá možnost byla vybraná. Pro otázky typu „text_field“ navrhuji zde ukládat asociativní pole[12], kde klíči budou jednotlivé textové odpovědi studentů, a hodnotami budou počty opakování těchto odpovědí. Jelikož MARAST se drží konvence, podle kterých označuje atributy používající JSON koncovkou „_json“, přidáme ji zde také.

Users_participated_json

Další nepovinný atribut typu **string**, kde pomocí formátu JSON navrhuji ukládat seznam uživatelů (studentů), kteří odeslali nějakou odpověď na dotyčnou otázku. Zde také pro zrychlení načtení a minimalizaci databázových dotazů, navrhuji použít agregaci. Tím se nahradí databázový dotaz, který by měl iterovat přes všechny odpovědi na otázku, aby detekoval všechny uživatele, kteří mají zaznamenanou nějakou odpověď na tuto otázku. Podle popsané konvence má atribut koncovku „_json“.

3.1.3 *PollAnswer*

Tuto tabulku navrhuji použít pro reprezentaci odpovědí uživatele na konkrétní otázku v anketě. I když v tabulce *PollQuestion* jsou agregovány výsledky otázek, pro implementaci požadovaných funkčností nestačí. Konkrétně bychom nemohli zjistit, jaké odpovědi vybíral konkrétní uživatel, což je jeden z funkčních požadavků na rozšíření. Proto jsou dále popsány atributy navrhované tabulky, která toto má umožnit.

[12] Asociativní pole je datový typ, který uchovává data ve formátu klíč-hodnota[23]

Poll_question_id, user_id

Jsou datového typu **integer** a jsou povinnými. Toto jsou cizí klíče vztažené k tabulkám PollQuestion a User, což znázorňuje to, že odpověď má obsahovat otázku, na kterou se odpovídá, a uživatele, který odpověď odeslal.

Kind

Je datového typu **enum** a je povinným, má stejnou hodnotu a smysl jak v tabulce PollQuestion. Přidat ho zde je výhodné proto, že budeme muset provádět méně databázových dotazů (nebude potřeba vždy hledat otázku, ke které patří odpověď), i když dojde k redundanci.

Result_json, text_field

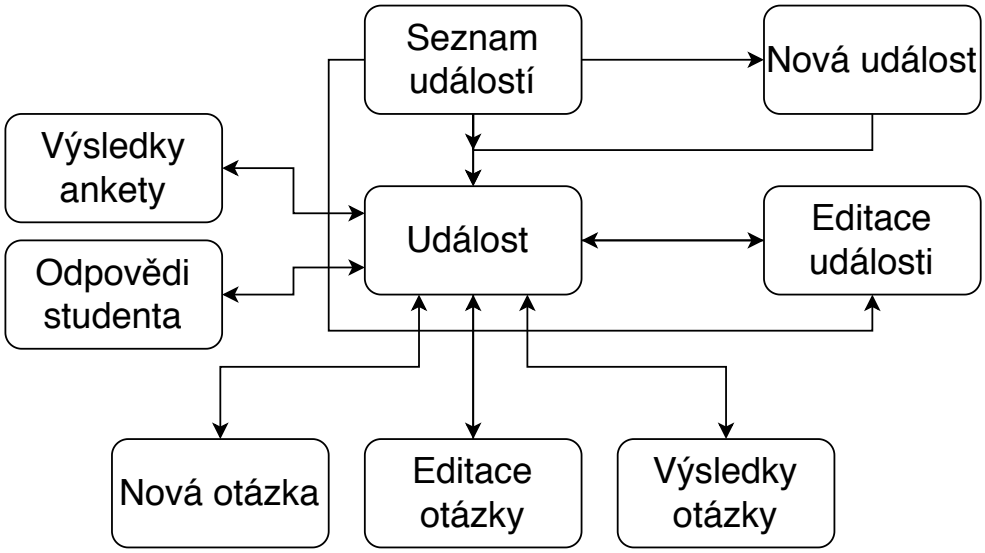
Jsou datového typu **string** a mají se použít pro ukládání obsahu odpovědi, žádný z nich není povinný. První, jak lze pochopit z názvu, ukládá data ve formátu JSON. Odpovědi typů „single_choice“ a „multi_choice“ navrhuji ukládat do atributu Result_json, a to ve formě pole, které obsahuje indexy vybraných možností odpovědí (např. první, třetí). Odpovědi na textové otázky naopak navrhuji ukládat jednoduše do atributu text_field. Tyto dva atributy jsem si rozhodl nesjednocovat do jednoho, protože tak to bude jednodušeji zpracovatelné během implementací, použití formátu JSON pro textové odpovědi práci jenom uměle zkomplikuje.

3.1.4 Závěr

Celkem, podle předchozího návrhu, kromě tří nových (navrhovaných) tabulek, budeme používat již existující tabulky *Semester, Course, User, Comment*. Navíc, jelikož pro nastavení uživatelských rolí (např. učitel, student), MARAST využívá tabulku Role, budeme používat i ji. Také, jelikož jedním z požadavků na toto rozšíření je možnost přidávání obrázků, použijeme existující v MARASTu model *Picture*.

3.2 Návrh pohledů, jako uživatelského rozhraní

V této kapitole provedu návrh jednotlivých stránek, jakož pohledů pro architekturu MVC. Při tomto návrhu máme přemýšlet o tom, aby systém byl snadno a pohodlně použitelný pro dvě docela odlišné skupiny uživatelů: vyučující, studenty. Toho lze docílit nejenom tím, že skupiny budou mít různé přístupné stránky, ale i tím, že obsah konkrétních stránek bude závislým na oprávnění uživatele. Toto, pro jednoduchost pochopení a snížení počtu vytvářených stránek, použijeme v uvedeném návrhu. I když se dá očekávat, že vyučující budou moci přistupovat k většímu počtu stránek než studenti.



Obrázek 3.1: Mapa webových stránek rozšíření

Hlavním cílem této části je vytvořit obecnou představu, co a v jaké formě budeme implementovat, jak se bude celý systém dělit na jednotlivé stránky. Není cílem předepsat konečnou podobu každé stránky, protože to se může v implementační částí změnit (můžeme pochopit, že co jsme vymysleli, uživatelsky přívětivým nebude). Dobrou zprávou ale je to, že změna pohledů (což jsou stránky, které uživatel vidí) vůbec neznamená, že budeme muset měnit řadiče a modely, což je nepochybná výhoda rozdělené architektury MVC.

Celkový návrh stránek pro toto rozšíření, a přechody mezi nimi, je znázorněn na obrázku 3.1, dále bude následovat popis všech navrhovaných stránek.

3.2.1 Seznam událostí

Tato stránka bude dostupná všem uživatelům MARASTu, odkaz na ni je plánováno umístit na horní navigační lištu. Ta ale bude jednou z těch stránek, o kterých bylo psáno dříve – bude měnit svůj obsah v závislosti na uživateli, který k ní v daný okamžik času přistupuje.

Pro studenty tato stránka bude představovat seznam událostí, ke kterým mají přístup (pravidla, podle kterých tento seznam bude vznikat, budou popsána v kapitole 4) a také odkazy na ně. Už ve fázi návrhu se dá předpokládat, že student může mít právo přístupu k velkému počtu událostí. Kupříkladu během doby výuky bude studentem velkého počtu předmětů, které událostí používají, navíc, k tomu se mohou přidat obecné události bez předmětu. Jelikož je tento stav možný a dokonce očekávaný, existuje možnost tuto potenciální nepohodlnost vyřešit.

Návrhem je rozdělovat seznam všech dostupných událostí na tři skupiny:

minulé, živé (aktivní) a budoucí. Toto částečně pomůže a zúží seznam událostí, ve kterých uživatel bude hledat jemu potřebnou. Úplně však problém nevyřeší – seznam obecně je příliš rozsáhlý, i když hledat v něm je teď o něco pohodlnější. Dalším možným řešením je přidat přepínač, který by filtroval celý seznam událostí podle vybraných předmětů a semestrů (toto by bylo obzvlášť užitečné pro studenty, kteří si jeden předmět zapisovali v několika semestrech, což na FITu není vzácný jev). Tato řešení dohromady mají vytvořit příjemné prostředí pro hledání nutné události.

Kromě pouhých názvů událostí by bylo dobré uvádět také kompaktní informaci o událostech, která by pomáhala s identifikací. Kupříkladu bychom zde mohli přidat zmíněný předmět a semestr, abstrakt, typ. Také určitě důležitým atributem, který by bylo dobré zařadit, je časový interval, ve kterém se událost bude uskutečňovat. Důležitým je například proto, aby studenti očekávali, že nebudou mít přístup k budoucí události.

Pro učitele tato stránka bude obsahovat více možností než pro studenty. Učitelé budou navíc také vidět odkazy na jiné stránky: vytvoření nové události, editace existující události, také budou mít možnost vymazání událostí. **Učitel by ale neměl možnost vymazávat úplně všechny události, jenom ty, které vytvořil on sám. Právo mazat všechno by měl mít jenom administrátor MARASTu**. Také po konzultaci s vedoucím práce bylo upřesněno, že **garant nějakého předmětu by měl mít právo vymazat všechny události patřící tomuto předmětu**. Uživatele, který tuto požadavky splňuje, budeme dále nazývat **oprávněným**.

V této fázi během konzultace s vedoucím práce bylo rozhodnuto, že učitelé budou mít přístup úplně ke všem událostem v systému, avšak nebudou je moci editovat nebo mazat (tyto práva budou mít jenom oprávnění uživatelé). Motivací pro to, abychom udělali všechny události přístupnými pro všechny vyučující je to, že v tomto případě vytvořené události mohou být testované, aby před začátkem události autor mohl dostat nějakou zpětnou vazbu od kolegů. Tuto zpětnou vazbu bude moci využít pro provedení nutných úprav v události, které ji zlepší.

Navzdory tomu, že učitelé budou mít přístup úplně ke všem událostem, na této konkrétní stránce je se seznamem událostí všechny nebudeme uvádět. V opačném případě (pokud bychom tak udělali) by byl tento seznam pro učitele příliš dlouhý, proto by v něm nikdo nic nehledal. Ale jelikož v aktuální verzi MARASTu role učitele platí pro celý předmět a ne pro konkrétní semestr, tak vyučující budou mít události patřící ke všem semestrům, ve kterých dotyčný předmět používá MARAST.

3.2.2 Nová událost

Jak bylo zmíněno v předchozí sekci (a znázorněno na obrázku 3.1), odkaz na tuto stránku budou mít učitelé, a to ze stránky se seznamem událostí toto bude jediným odkazem. Zde uživatelé budou moci vybrat atributy pro konkrétní

typ události, které byly popsané v kapitole 3.1. Zásadním atributem je typ události proto, že v závislosti na něm, budeme nabízet různá pole pro vyplnění (kupříkladu nemá smysl prosit uživatele vybírat, v jakém pořadí se budou zobrazovat komentáře k události typu Anketa, protože možnost komentování u tohoto typu události neplánujeme). To, jakým způsobem tento rozdíl bude řešen, necháme na implementační fázi. Po konzultaci s vedoucím práce bylo rozhodnuto, že v případě anket, komplikovat tuto stránku vytvářením a editací anketních otázek by bylo nesprávné, počet polí pro vyplnění by byl příliš velký, vytvářelo by to nepříjemný dojem pro uživatele. Proto pro vytváření a editací otázek uděláme samostatné stránky, které budou popsané dále.

Po potvrzení uživatelem a poslání požadavku na vytvoření nové události serveru, má být učitel přesměrován na nově vytvořenou událost, aby zkontroloval, že všechno je podle jeho představ.

3.2.3 Událost

Toto je centrální stránka celého rozšíření, na ní bude vidět větší část provedené práce. Tato stránka bude dostupná učitelům i studentům, ty jí ale budou vidět jinak. Každý bude vidět tuto stránku jinak v závislosti na tom, jakého typu událost je. Na začátku popíšeme to, jak bude stránka vypadat pro studenta v závislosti na typu události.

Jak bylo zmíněno v kapitole 3.1, oba dva typy událostí mají množinu společných atributů, které by bylo vhodné pro konzistenci zobrazovat podobným způsobem. V této množině, kromě typu, jsou název, autor, začátek, konec, semestr, předmět, abstrakt, tělo, anonymní odpovědi (anonymous_responses). Budeme je zobrazovat ve formě tabulky v hlavičce stránky. Dále pod tuto tabulku umístíme funkčnosti samotné, které máme implementovat.

Pro Q&A události tady bude seznam otázek, odpovědí a obyčejných komentářů. Během konzultací s vedoucím práce bylo rozhodnuto rozdělit seznam komentářů (také otázek a odpovědí) na dvě skupiny: aktivní, neaktivní. To pomůže studentům se lépe orientovat v už existujících příspěvcích a nepsat opakovaně komentáře se stejným smyslem, a místo toho hlasovat o komentáře, v nichž je již jejich poznámka obsažena. Učitelům to dovolí jednodušeji vybírat otázky, na které následně odpovídají (jednodušší to bude díky tomu, že budou vybírat z menšího počtu otázek).

Pro ankety, na místě otázek a odpovědí, budou anketní otázky s možnostmi odpovědí (v případě *singlechoice* a *multichoise*), nebo s textovým políčkem (v případě textfield) určené k tomu , aby je studenti vyplnili. Otázky, stejně jako komentáře, mohou být vyplněny jedna za druhou shora dolů. Příliš hodně těchto otázek zde být nemůže, protože nejsou určené k tomu, aby studenti věnovali celý čas hodiny jejich vyplňování, k takovým účelům lze mnohem lépe použít jiné možnosti MARASTu (kupříkladu kvíz), což se dokonce využívá pro provedení některých zkoušek na FITu. Proto dělit tyto anketní otázky na nějaké sekce není potřeba.

Pro učitele stránka událostí bude vypadat velmi podobně. Rozdílem bude to, že na ní bude více dostupných funkčností. Společným pro oba typy událostí bude to, že učitelé budou mít odkazy na stánky s editací a s možností smazat událost z databáze.

Pro události typu Q&A bude i možnost manuálně měnit pořadí komentářů, možnost označit komentář jako zodpovězený, a tím převést do skupiny neaktivních (a naopak).

Pro události typu Anketa budou moci oprávnění učitelé odsud přejít na stránky pro vytváření, editaci, výsledky anket. U událostí, které nemají nastavenou anonymitu, také na stránku s odpověďmi konkrétního studenta.

3.2.4 Editace události

V této sekci oprávnění uživatelé budou moci měnit parametry události. Odkaz na ni, bude dostupný učitelům v seznamu událostí a také v samotné události. I když takové odkazy mohou částečně přeplnit stránku se seznamem (protože nejspíše budou realizované formou tlačítek), pomůže to zkušenějším uživatelům nenavštěvovat nepotřebnou stránku (událost), což šetří čas, zmenší počet požadavků na server a také zlepší celkový dojem u uživatele.

Po úspěšné editaci uživatel má být přesměrován na související událost, aby zkontroloval, zda to, co provedl, bylo tím, čeho chtěl docílit. Jiné chování, kupříkladu přesměrování na seznam událostí, by bylo zvláštní a ne moc uživatelsky přívětivě, také zvolené řešení je konzistentní s vytvářením nové události.

3.2.5 Nová otázka

Odkaz na tuto stránku má být dostupen oprávněným učitelům ze stránky s událostí typu Anketa. Tady uživatele budou moci vytvářet tři, popsáné dříve, typy otázek: *singlechoice*, *multichoice*, textová. Po vytvoření otázky uživatel by měl být přesměrován na stránku se související událostí. Jelikož vytvoření chceme provádět takto na speciální stránce by bylo dobře přidat kromě klasického tlačítka „Vytvořit“, ještě jedno, které by se nazývalo, kupříkladu, „Ještě jedna otázka“, které by zlepšilo uživatelskou přívětivost v případě, že uživatel chce vytvořit rovnou několik otázek.

3.2.6 Editace otázky

Odkaz na tuto stránku bude umístěn vedle každé otázky v událostech typu Anketa, zde uživatelé budou moci měnit zadání a možnosti odpovědí na otázku. Po změně pole, které chce změnit uživatel, a odeslání požadavku na server, by měl být uživatel přesměrován zpět na stránku s událostí.

3.2.7 Výsledky otázky, výsledky ankety

Odkazy na tyto stránky budou dostupné na stránce se související událostí. To, zda výsledky všech otázek patřící události mají být uvedené na jedné stránce nebo na samostatných stránkách ve fázi návrhu, těžko říct, oba tyto nápady mají své výhody i nevýhody.

Výhodou umístění na jednu stránku je to, že uživatelé budou mít globální přehled všech otázek, z čehož jednodušeji budou moci dospět k nějakému obecnému závěru, který platí pro (skoro) všechny otázky. Nevýhodou je, že v tomto případě bude pro uživatele složitější se soustředit na nějaké konkrétní otázky.

V případě rozdělení na samostatné stránky jsou popsané výhody a nevýhody opačné. Další výhodou tohoto přístupu je rychlejší načítání stránky, neboť jednoduše pro ni bude potřeba méně dat ze serveru.

Kvůli tomu se během implementace pokusíme implementovat obě možnosti, ze kterých potom vybereme vhodnější.

3.2.8 Odpovědi studenta

Odkaz na stránky s událostí typu Anketa budeme přidávat, nebo ne, v závislosti na tom, zda u této události je nastaveno, že má anonymní odpovědi (dostupné to bude jenom oprávněným uživatelům). Tímto splníme požadavky číslo 6 a 7 ze sekce 2.5.3.

Na této stránce učitel bude moci vybrat studenta, jehož odpovědi si chce prohlížet. Dělat tady samostatnou stránku pro každou odpověď studenta by nebylo uživatelsky přívětivé, proto na této stránce budou všechny odpovědi studenta dohromady. Pro pohodlnost použití by tady měl být nějakým způsobem přidán odkaz zpět na událost.

V této fázi bychom si měli vytvořit představu, jak tato stránka bude vypadat. Logicky je celá tato stránka rozdělená na dvě části:

1. Výběr studenta, jehož odpovědi na otázky chceme zkoumat

2. Prohlížení odpovědí vybraného studenta

Návrhem je, pro jednoduchost následné implementace, tuto stránku rozdělit na dvě, z nichž každá bude zodpovědná za svoji logickou část (výběr studenta nebo prohlížení jeho výsledků).

Výběr studenta můžeme udělat formou seznamu studentů, kteří odpověděli na nějakou otázku v události typu Anketa. U každého studenta by byl odkaz vedoucí na stránku s prohlížením jeho výsledků.

Prohlížení můžeme realizovat tím způsobem, že nasimulujeme to, jak tyto otázky vidí student. U otázek, na které student odpověděl, jeho odpovědi vypíšeme. U otázek, na které ještě neodpověděl, napíšeme, že odpověď není.

3.3 Návrh řadičů

V této kapitole provedeme návrh poslední části rozdělené architektury MVC – řadičů. Jelikož už máme navrženy dvě ostatní komponenty MVC, úkolem, který zde budeme řešit, je to, abychom pomocí navržených modelů mohli zprovoznit (naplnit daty) navržené pohledy. Je to i úkol, který řadiče mají řešit. Jelikož v architektuře MVC jednomu modelu klasicky odpovídá jeden řadič, celkem vytvoříme 3 nové řadiče, které v této kapitole navrhneme.

Obsah řadičů se dělí na metody (akce), které provádějí nad databází naprogramované operace. *Ruby on Rails* používá konvenci pro výchozí „klasické“ metody pro každý řadič, které následně můžeme měnit podle požadavků na aplikaci. Celkem je těchto metod 7, každá z nich má řešit „klasické“ úlohy. Mezi těmito metodami jsou: metoda pro vykreslení stránky s vytvářecím formulářem (*new*), s editačním formulářem (*edit*), se seznamem všech záznamů v konkrétní tabulce (*index*), s jedním záznamem (*show*), metoda pro zápis dalšího záznamu do tabulky (*create*), metoda pro změnu konkrétního záznamu (*update*), metoda pro vymazání konkrétního záznamu (*delete*). Jak už bylo zmíněno dříve, nemusíme používat všechny této metody, a dokonce můžeme měnit jejich smysl, což se ale nedoporučuje. Také k tomuto seznamu můžeme přidávat i další metody.

Cílem této kapitoly je navrhnut co a jaká metoda, v jakém řadiči řeší. Zatím nedefinujeme, jak přesně to má dělat, toto necháme na implementační fázi. V této kapitole se budeme hodně odkazovat na kapitolu 3.2, protože v ní jsme definovali, jak mají vypadat stránky, které chceme zprovoznit.

Dále bude uveden návrh metod (akcí) tří našich řadičů. Samotné řadiče, podle jiné konvence, popsané v sekci 2.2.2, se budou nazývat: *EventsController*, *PollQuestionsController*, *PollAnswersController*.

3.3.1 *EventsController*

Řadič pro události. Návrhem je, že bude obsahovat všech 7 „klasických“ metod a 3 další, které budou obsluhovat 3 stránky, které budou specifické pro náš úkol. Dále bude následovat stručný návrh obsahu těchto metod.

Index

Tato metoda bude zcela naplňovat daty webovou stránku se seznamem událostí (popsaná v sekci 3.2.1). V závislosti na uživateli, který na ni přistupuje, bude sestavovat seznam jeho událostí. Jelikož plánujeme mít na této stránce přepínač podle semestrů a předmětů (který bude filtrovat události), máme jeho správné fungování nastavit právě zde v řadiči, ne v pohledu, protože toto řešení by nesplňovalo principy architektury MVC (jak je popsáno v sekci 2.2.1 pohled má především zobrazovat data, ne nějakým způsobem je zpracovávat).

Ze stejného důvodu zde máme rozdělovat události na tři skupiny, popsané v sekci 3.2.1: minulé, aktivní a budoucí.

Show

Tato metoda bude zodpovědná za stránku s konkrétní událostí (popsaná v sekci 3.2.3). Její prací bude vyhledat potřebnou událost a předat ji do pohledu, kde se bude zobrazovat. Jelikož spolu s událostí předáme i její typ (je to jeden z atributů tabulky Event), pohled bude moci podle typu požádat další data k zobrazení – komentáře pro událost typu Q&A nebo anketní otázky pro anketu. Jelikož pohled bude jenom spouštět metody v řadičích, toto řešení je validní a neporušuje principy MVC.

New

Tato metoda bude zodpovědná za zobrazení stránky s formulářem pro novou událost (popsaná v sekci 3.2.2). To, jak vyřešíme problém zmíněný v této sekci (abychom nenutili uživatele vyplňovat zbytečné atributy), necháme na implementační fázi.

Create

Metoda bude zodpovědná za obsloužení požadavku, který bude pocházet ze stránky s vytvářecím formulářem. Jejím hlavním cílem bude vytvořit novou událost v databázi, případně oznámit uživateli, co ve vytvářecím formuláři vyplnil špatně (např. že nevyplnil povinný atribut). Jak je zmíněno v sekci sekci 3.2.2, po úspěšném vytvoření události má uživatel být přesměrován do nově vytvořené události.

Edit

Zobrazí formulář pro editaci události. Pro pohodlnost použití má zobrazovat už vybrané (aktuální) hodnoty atributů.

Update

Tato metoda bude aktualizovat záznam o události v databázi. Stejně jako metoda *create* má dávat uživatelům najevo, že nové atributy (které teď chtějí uložit místo starých) nejsou validní. Po úspěšné editaci také má přesměrovat uživatele na stránku s událostí (zmíněno v sekci 3.2.6).

Delete

Smaže záznam o konkrétní události z databáze, potom přesměruje uživatele na seznam všech dostupných událostí.

Results

Tato metoda bude obsluhovat stránku s agregovanými odpověďmi všech studentů na všechny otázky v anketě (popsaná v sekci 3.2.7).

Select_student

Táto metoda bude obsluhovat stránku s výběrem konkrétního uživatele (kterou jsme popsali v sekci 3.2.8), jehož odpovědi chceme prohlížet. Úkolem této metody bude sestavit seznam uživatelů, kteří odpověděli aspoň na nějakou otázku z ankety a předat tento seznam do pohledu.

Results_of_user

Tato metoda bude obsluhovat stránku s odpověďmi konkrétního uživatele na otázky v anketě (popsána v sekci 3.2.8).

3.3.2 *PollQuestionsController*

Řadič pro otázky v anketě. Návrhem je využití 6 „klasických“ metod (všech kromě indexu) a jedné specifické právě pro tento řadič. Uvádět zde popis všech „klasických“ metod by bylo zbytečné, protože mají stejné cíle, jako související metody v *EventsController*, které byly popsané v předchozí sekci. Navíc to, kde najdeme odkazy na stránky, za které jsou tyto metody zodpovědné, jsme uváděli v kapitole 3.2. Proto zde popíšeme jenom jednu z „klasických“ metod, protože u ní není zřejmě, na co se využije (jedná se o metodu show). Také uvedeme popis specifické metody pro tento řadič, o které bylo zmíněno na začátku odstavce.

Show

Tuto metodu podle návrhu použijeme pro poskytování dat ze stránky s výsledky konkrétní otázky v anketě. Máme tuto metodu zatím nevyužitou, protože ji nebudeme používat pro zobrazení otázek v události.

Show_result

Tato metoda je specifickou pro uvedený řadič. Má sloužit jako samostatný zdroj agregovaných výsledků konkrétní otázky. Přidáváme ji pro zrychlení procesu zobrazení výsledků.

3.3.3 *PollAnswersController*

Řadič pro odpovědi na otázky v anketě. Ten bude mít podle návrhu pouze jedinou metodu – *create*. Žádné další naplánované nejsou ze dvou důvodů. První je „praktický“a spočívá v tom, že nepotřebujeme speciální stránku pro

vytvářecí formulář, protože ten bude těsně svázán s otázkou, na kterou chceme vytvořit odpověď (to samé platí pro stránku se seznamem odpovědí). Druhý důvod je „politický“ a spočívá v tom, že nechceme nikomu umožnit mazat a měnit odpovědi po jejich odeslání. To by několikrát znehodnotilo výsledky, které učitelé dostávají od studentů.

Create

Smysl této metody je stejný jako v sekci 3.3.1 – vytvoří novou odpověď. Jedinou zvláštností je to, že před vytvořením odpovědi máme zkontrolovat, aby uživatel na dotyčnou otázku ještě neodpovídal.

Kapitola 4

Implementace

V této kapitole budou popsané technologie, postupy a rozšíření frameworku, které jsem použil během implementace této bakalářské práce. Testování implementovaných funkčností bude věnovaná celá kapitola 5.

Před tím, než přejdeme k popisu toho, jak byly implementované konkrétní stránky (které jsme si navrhli v sekci 3.2), máme vytvořit modely, jakož základ, ze kterého začneme (ty jsme navrhli v sekci 3.1)

4.1 Databáze

Pro vytvoření modelů (tabulek) *Ruby on Rails* nabízí příkazy spustitelné z terminálu. Prvním s těchto příkazů je

```
rails g model Event atribut1:type atribut2:type
```

V tomto příkazu popisujeme model, který chceme vytvořit. Místo *Event* můžeme napsat libovolný jiný název modelu. Píšeme ho v jednotném čísle. Tento příkaz nemění databázi přímo, jen provádí nutné přípravy. Jednou z věcí, kterou provádí je to, že vytvoří tzv. migraci.

Smyslem takových migrací je, že dovolují postupně měnit strukturu databáze [13]. Kromě toho, že migrace obsahují požadované změny v databázi, obsahují také i čas jejich vytvoření. Všechny takové migrace, které byly vytvořeny během celého vývoje projektu, MARAST skladuje v jedné složce, což dovoluje mít celou historii změn v databázi.

Abychom mohli migraci spustit a konečně provést změnu v databázi, použijeme příkaz

```
rails db:migrate
```

Vztahy mezi tabulkami

Důležité je, jakým způsobem vytvoříme vztahy mezi tabulkami. Proto, abychom k tabulce přidali cizí klíč, v příkazu pro vytváření modelu použijeme klíčové

slovo „references“ a to ve formátu jméno_modelu (na který se chceme odkazovat): references. Pokud například chceme v tabulce Event vytvořit cizí klíč vztažený k modelu *Semester*, použijeme následující příkaz:

```
rails g model Event semester:references
```

Jelikož v našem návrhu jsme použili vazby typu 1:N, uvedeme to, jakým způsobem postupujeme dále při jejich implementaci. Na straně odpovídající tabulce, která bude odpovídat „1“ (id záznamů které budou představovat cizí klíče), v modelu přidáme řádek:

```
has_many :events, dependent: :destroy
```

„Dependent: :destroy“ nemusí zde nemusí být povinně a znamená to, že pokud se smaže záznam v tabulce, mají se smazat i všechny záznamy v druhé tabulce, které se na něho odkazují.

Na straně odpovídající „N“, v modelu přidáme:

```
belongs_to :semester
```

Užitečným typem vazeb, které byly použity během implementace, je polymorfická vazba.

Polymorfická vazba

V *Ruby on Rails*, pomocí polymorfických vazeb mezi modely, můžeme docílit toho, že se jeden model přes jeden cizí klíč může odkazovat rovnou na několik jiných modelů [24]. Takový typ vazby MARAST využívá pro modely *Comment* (komentář) a *Picture* (obrázek). Jak bylo uvedeno v sekci 3.1, tyto modely použijeme v naší implementaci.

U obou zmíněných modelů se využívá pro nastavení toho, k čemu se vztahují. Kupříkladu komentář může být přidán pod příspěvek v blogu nebo také k příkladu v cvičebnici. Obrázek může patřit k lekci nebo také k příspěvku v blogu.

Jelikož tyto modely před začátkem práce již existovaly, stačilo jenom nastavit jejich vazby, abychom je mohli použít v rámci modelu *Event* (událost).

Pro nastavení polymorfické vazby ji musíme registrovat na obou stranách. Jako příklad zde uvedeme příkazy nutné pro vytvoření (polymorfické) vazby mezi modely *Event* a *Comment*.

V modelu komentáře použijeme příkaz (ten už byl před začátkem práce):

```
belongs_to :commentable, polymorphic: true
```

Tím nastavíme, že budeme používat polymorfickou vazbu a že pro odkazování na model (který neznáme), budeme používat jméno „commentable“.

Potom v modelu *Event* nastavíme, že chceme vystupovat v roli „commentable“. To uděláme příkazem:

```
has_many :comments, as: :commentable,
                    dependent: :destroy
```

Vazbu poté využijeme kupříkladu takto (jednoduchá ukázka):

```
event = comment.commentable
```

Povinné atributy

Pro nastavení toho, že atributy jsou povinné v odpovídajícím modelu doplníme speciální příkaz, kde uvedeme seznam těchto atributů. Například pro model Event vypadá takto:

```
validates :user,
          :title,
          :abstract,
          :form_of_study,
          :kind,
          presence: true
```

Jeho cílem je udělat autora, název, abstrakt, formu studia a typ povinnými atributy.

Enum (výčtový typ)

Enum (výčtový typ) v databázi budeme ukládat ve formě celého čísla (integer). V odpovídajícím modelu následně každému využitému identifikátoru (číslu) přiřadíme hodnotu – textový řetězec. Uděláme to pomocí příkazu ve formě „enum název_atributu: { hodnota1: identifikátor1, hodnota2: identifikátor2 }“. Pro příklad, uvedeme to, jak tento příkaz reálně použijeme pro nastavení atributu kind (typ) v modelu Event:

```
enum kind: { q_and_a: 0, poll: 1 }
```

4.2 Kompletní implementace pohledů

Jelikož celé rozšíření má být přístupné přes uživatelské rozhraní, které je přístupné přes internetové stránky, implementaci popíšeme právě v souvislosti s každou naplánovanou stránkou (které jsme si navrhli v sekci 3.2).

Navigační lišta

Tvorba uživatelského rozhraní se začala tím, že byl přidán odkaz na stránku se seznamem událostí (sekce 3.2.1) na navigační lištu. V této fázi jsme použili to, že MARAST využívá rozdělení stránek na takzvané *partials* (podpohledy [1]). Smyslem jejich využití je, abychom minimalizovali opakování stejného kódu, konkrétně ty pomáhají daný úkol vyřešit v rámci pohledů. Jedná se o aplikování principu DRY (zmíněno v sekci 2.2.1). Existuje konvence pojmenování podpohledů, ta spočívá v tom, že na začátek jejich názvů přidáme znak „_“,

této konvence se budeme držet během implementace. Podpohledy fungují tak, že definujeme části pohledů, které potom využijeme pro vytvoření více stránek. Jednou z takových části byla horní navigační lišta, u které stačilo jenom přidat jedno tlačítko s odkazem na seznam událostí. Tato navigační lišta se potom automatický použije na všechny stránky v rámci MARASTu.

4.2.1 Seznam událostí

První stránkou, která byla vytvořena, byla stránka se seznamem událostí přístupných uživateli. Podle návrhu (3.2.1), kromě samotného seznamu událostí, zde má být přepínač, který bude filtrovat události podle vybraných semestrů a předmětů. Také máme rozdělovat události podle toho, zda jsou minulé, aktivní nebo budoucí. Na začátku máme sestavit zmíněný seznam.

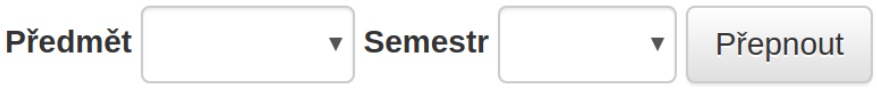
Jelikož je v seznamu požadavků (2.5) uvedeno, že události máme rozlišovat podle forem studia, toto uplatníme i zde. Událost přidáme do seznamu přístupných uživateli jen tehdy, pokud souvisejí formy studia uživatele (vztažené ke konkrétnímu semestru a předmětu) s událostí. „Souvisejí“ v tomto případě znamená to, že se buď shodují, nebo aspoň jedna z nich je neurčená. Toto je jedna ze dvou nutných podmínek pro přidání do seznamu.

Sestavení seznamu přístupných událostí

Pro řešení tohoto úkolu byla vytvořena další metoda v modelu *User* (tam logicky patří nejlíp) – **events_of_user** (události uživatele).

Jelikož v seznamu požadavků (sekce 2.5) , kromě samotného seznamu událostí, zde má být přepínač, který bude filtrovat události podle vybraných semestrů a předmětů. Také máme rozdělovat události podle toho, zda jsou minulé, aktivní nebo budoucí. Na začátku máme sestavit zmíněný seznam. „Souvisejí“ v tomto případě znamená to, že se buď shodují, nebo aspoň jedna z nich je neurčená. Toto je jedna ze dvou podmínek pro přidání do seznamu.

Druhá spočívá v tom, aby souvisely semestr a předmět události a semestr a předmět nějaké role uživatele (popsáno v sekci 2.3.1).



Obrázek 4.1: Přepínač semestrů a předmětů

Výběr semestru a předmětu

Tento požadavek splníme tak, že vytvoříme dvě rozbalovací nabídky (**<select> tag**), ve kterých uživatelé budou vybírat ze seznamů předmětů a semestrů, ve kterých mají aspoň nějakou roli (nebylo by uživatelsky přívětivě uživatele nutit

k tomu, aby vybírali úplně ze všech semestrů a předmětů). Ve výchozím stavu v těchto nabídkách nebudou vybrány žádný semestr a předmět, bude uveden seznam úplně všech událostí přístupných uživateli. Jak to vypadá, lze vidět na obrázku 4.1. Proto, abychom mohli rychle načítat seznamy předmětů a semestrů, jsem přidal dvě metody do modelu User (uživatel), kam logicky patří nejlépe: jedna pro semestry, druhá pro předměty. Také přidáme tlačítko „Přepnout“, které aktualizuje seznam událostí v závislosti na tom, jaký semestr a předmět vybral uživatel. Uděláme to tak, že přidáme vybraný semestr a předmět (jejích id) jako parametry v odkazu. Při vykreslení nové stránky je už v řadiči použijeme pro filtraci v seznamu všech událostí uživatele. Také předané argumenty použijeme pro vyplnění výchozích hodnot v rozbalovacích nabídkách, jinak by to nebylo přívětivé pro uživatele.

Rozdělení na minulé, aktivní, budoucí

V této fázi v řadiči máme relevantní seznam událostí, ty máme rozdělit na tři skupiny a předat pohledu, aby je zobrazil. V tomto rozdělení nám pomůže příkaz:

```
Time.zone.now
```

Ten nám vrátí aktuální čas, který budeme moci porovnávat s uvedenými časy začátku a konce událostí. A to jednoduše pomocí operátorů >, <, >=, <=. Uděláme to tak, že pokud událost nemá ani jeden z časů uvedený, bude to znamenat, že je aktivní vždy.

Po tomto rozdělení skupiny předáme do pohledu ve formě tří proměnných.

Samotný pohled

Po přidání výběru semestrů a předmětů jsem pochopil, že užitečným by bylo přidat tlačítko, které by vracelo stránku k původní podobě, kde není vybraný žádný semestr a předmět. To jsem přidal a nazval „Všechny moje události“.

Posledním úkolem, který v této fázi zbylo vyřešit, je to, jakým způsobem zobrazovat odkazy na samotné události. Ten jsem vyřešil tak, že jsem vytvořil *partial*, který je zodpovědný za zobrazení seznamu událostí, které dostává jako lokální proměnnou. Tento *partial* zde využijeme třikrát (pro každou ze skupin událostí). Pro každou událost zobrazíme všechny informace popsané v sekci 3.2.1 informace. Také, pro oprávněné uživatele, přidáme odkaz na editací události, možnost smazání události.

4.2.2 Nová událost

Tuto stránku jsme navrhli v sekci 3.2.2. Už během návrhu jsme pochopili, že budeme muset tuto stránku měnit v závislosti na typu události. To implementujeme tak, že celý proces vytváření události rozdělíme na dvě stránky:

1. Výběr typu události

2. Výběr ostatních atributů

Tímto způsobem původně naplánovanou stránku zjednodušíme. Navíc se jedná o řešení konzistentní s ostatními částmi MARASTu. Příklady v MARASTu se vytvářejí úplně podle stejného postupu (na první stránce uživatel vybírá typ vytvářeného příkladu, na druhé nutné parametry pro tento typ).

Obsluhovat tuto stránku budou dvě klasické metody v řadiči: *new* a *create*. První z nich bude sloužit pro vykreslování obou stránek vytvoření události. Druhá pro samotné přidání záznamu do databáze. Zajímavým je, jakým způsobem zobrazujeme stránky pro vytvoření události, to zde uvedeme.

Pohledy

Byl využit jenom jeden pohled, obsah kterého se ale značně mění v závislosti na tom, zda už byl vybrán typ události nebo ne. Typ je předáván jako parametr v odkazu, proto lze docela jednoduše zkontrolovat, zda už nějaký byl vybrán. Na začátku uživatel žádný typ v odkazu nemá, proto se zobrazuje stránka s výběrem typu. Zobrazuje se pomocí už dříve představených podpohledů. Pro tuto stránku jsem vytvořil dva – pro každý krok vytvoření události jeden.

4.2.3 Událost

Jak bylo zmíněno v návrhu (3.2.3), v hlavičce stránky chceme vytvořit tabulku s parametry události. Toto jednoduše uděláme pomocí HTML prvku „table“ (tabulka), u kterého za účelem konzistence použijeme styl ostatních tabulek v MARASTu (styly v MARASTu jsou popsané v sekci 2.3.1).

Jelikož se tato stránka bude významně lišit v závislosti na typu události, následující povídku rozdělíme na dvě podkapitoly.

4.2.3.1 Q&A

Jak bylo zmíněno v sekci 3.1, pro příspěvky, které uživatelé mohou přidávat k událostem typu Q&A, jsem se rozhodl využít existující upravený model *Comment* (komentář), který je popsán v sekci 2.3.

Komentář

Před začátkem práce se tento model používal na více místech MARASTu. Ten model ale kromě toho, že nepodporoval anonymní komentáře, měl ještě další omezení, která pro požadované rozšíření nebyla vhodná. Proto jsem tento model upravil. Provedené úpravy budou popsané dále.

Hlasování

První změna, která byla provedena, spočívala v tom, že bylo umožněno hlasování nejenom o příspěvky typu odpověď, ale i o obecné komentáře a otázky. Pro tuto změnu bylo potřeba upravit nejenom pohledy (aby se tato možnost vůbec zobrazovala), ale i tzv. „validátor“, což je speciální třída, jejímž úkolem je rozhodnout, zda můžeme nějakou akci provést, nebo ne (podobný princip se využívá pro přístupová práva popsaná v 2.3.1).

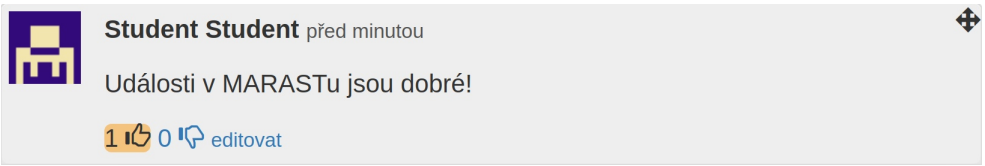
Automatické řazení

Potom bylo přidáno řazení všech druhů komentářů podle času a podle hlasů. Toto bylo realizováno přidáním dvou dalších metod do modelu komentář (kde bylo implementované původní řazení). Učitel pak toto chování nastavuje v editaci události pomocí atributu *mod_of_inquiries*. Tímto splníme požadavek č. 3 z 2.5.2.

Manuální řazení

Možnost pro učitele manuálně řadit otázky (požadavek č. 4 z 2.5.2) implementujeme podle principu, který v MARASTU již existoval. Ten je používán v pohledu pro vytvoření symbolického příkladu. Abychom ho použili i ve svém rozšíření, stačilo pouze upravit podpohled, který je zodpovědný za zobrazení seznamu komentářů, které přidáme jako lokální proměnnou. Konkrétně šlo o použití speciální třídy CSS a následně o doladění stylů, aby se komentáře v rámci vytvářeného rozšíření nelišily od ostatních.

Na začátku byla přidaná možnost přesouvání úplně všech komentářů, to ale působilo chaoticky, zvláště když se přesouval komentář, který je vztažený k nějakému jinému komentáři (komentáře se dají vnořovat). Proto přesouvat povolíme jenom aktivní komentáře, protože u těch neaktivních to smysl nemá – na ty už učitel odpověděl. Navíc se omezíme na ty, které jsou na první úrovni (nejsou vztažené k nějakému dalšímu komentáři).



Obrázek 4.2: Příklad komentáře

To, že uživatel může komentář přesouvat, označíme speciálním znakem v horním pravém rohu komentáře (viz obrázek 4.2).

Jelikož se jedná jenom o lokální změnu, toto přesouvání povolíme úplně všem učitelům.

Anonymní dotazy

Toto chování učitelé nastavuji v editaci, resp. vytvoření události. Na začátku to bylo implementováno s logikou, že pokud toto je vybráno, tak úplně všechny komentáře v diskuzi budou anonymní. Potom se ale zjistilo, že to působí velmi „anarchicky“, proto bylo rozhodnuto, že komentáře učitelů anonymními nebudou ani v tomto případě. Navíc se takové řešení využije pro funkční požadavky č. 5 a 6 z 2.5.2 (v nich je požadováno, aby učitelé mohli textově odpovídat na otázky a také přidávat odkaz na čas, kde se odpovědi nacházejí ve videu). Jinak by nebylo možné odlišit komentáře učitelů od ostatních účastníků diskuze.

Implementováno bylo úpravou podpohledu zodpovědného za zobrazení konkrétního komentáře. V případě, že u události jsou zapnuté anonymní dotazy, nebudeme zobrazovat jméno a fotografii uživatele. Stejně ale do databáze id autora komentáře ukládat budeme. I když by se jednoduše dalo zrušit povinnost atributu autor, toto dělat nebudeme. V tomto případě bychom nemohli detekovat autora komentáře, což by znamenalo, že máme možnosti dvě:

1. Nedovolit studentům měnit a mazat žádné komentáře vůbec.
2. Dovolit každému měnit a mazat každý komentář.

Ani jedna z těchto možností není ani uživatelsky přívětivá, ani vhodná. Tohle řešení bude dostačujícím, jelikož učitel MARASTu nemá nyní možnost zjistit, kdo jak odpovídal, což je to, čeho bychom chtěli docílit. Také toto řešení má výhodu v tom, že měníme model co nejméně.

U anonymních dotazů bylo zrušeno omezení, že uživatel nemůže hlasovat o své komentáře, což zlepší pocit anonymity pro studenty. Pro toto bylo potřeba změnit *policy* soubor definující práva na spouštění akcí v řadiči.

Nekontaktní varianta – možnost textové odpovědi na otázky

Tento funkční požadavek splníme triviálně tím, že používáme komentáře v MARASTu.

Možnost pro učitele snadno zpětně přidat odkaz na konkrétní čas ve videu s odpovědí

Záznamy předmětů vyučovaných v MARASTu jsou uložené a přístupné na platformě *YouTube*[13]. Ta podporuje generování odkazu na videa, u kterých lze nastavit čas začátku. Podle stavu k roku 2020, aktuální verze platformy *YouTube* to poskytuje pomocí tlačítka v menu, které se otevře stisknutím pravého tlačítka myši na video. U komentářů v MARASTu lze používat značkovací jazyk *Markdown*, který poskytuje jednoduchý způsob, jak lze přidávat

[13] https://www.youtube.com

odkazy, u kterých lze nastavit to, co na odkazu bude napsáno. Dělá se to pomocí syntaxe:

```
[popis textem](odkaz na YouTube s parametrem)
```

O této možnosti učitelům dáme vědět na stránkách pro editaci a vytvoření události.

Stav komentáře

Posledními funkčními požadavky na Q&A události, které jsme měli vyřešit, byly „Možnost zvýraznění právě zodpovídané otázky“ a „Rozdělení otázek na aktivní a neaktivní“. Oba požadavky implementujeme pomocí vhodné úpravy modelu *Comment*. Touto úpravou je přidání nového atributu **status**, , který je datového typu **enum** a může nabývat hodnot: „aktivní“, „zvýrazněný“ a „neaktivní“. Výchozím stavem je aktivní. Pomocí tohoto atributu oddělujeme neaktivní příspěvky od ostatních, abychom je mohli zobrazovat zvlášť. Také pomocí něho můžeme komentáře zvýrazňovat, a to změnou barvy komentáře na červenou (žádné jiné příspěvky tuto barvu nemají).

Abychom dovolili učitelům měnit stav komentáře, byl upraven proces editace komentáře. V rámci události učitelé mohou měnit nejenom své komentáře, ale i jiných uživatelů, ač jen omezeně. Učitelé vidí tlačítko „editovat“ u všech komentářů a po jeho stisknuti vidí formulář pro editaci. Ten formulář v případě, že učitel není autorem tohoto komentáře, obsahuje jenom stav. Tímto omezení implementujeme na úrovni pohledů. Také toto omezení implementujeme v řadiči a to tím způsobem, že budeme v metodě *update* ignorovat tělo komentáře.

Studenti stav komentářů měnit nemohou.

4.2.3.2 Anketa

Podle našeho návrhu v kapitole 3.1, pro otázky v anketě použijeme model *PollQuestion*; pro odpovědi model *PollAnswer*. Dále bude popsáno, jakým způsobem je budeme používat. Hlavním úkolem, který řešíme, je zobrazení otázek a možných odpovědí na ně. Zároveň, když uživatel na otázku již odpověděl, chceme, aby se mu jeho odpověď zobrazovala.

Jelikož v aktuální verzi rozšíření patří každá otázka vždy jedné události, otázky související s událostí najdeme jednoduše pomocí dotazu na model.

Tyto otázky předáme do podpohledu „_show_many“, který je zodpovědný za zobrazení seznamu otázek. Ten každou otázku předá do dalšího podpohledu, který už je zodpovědný za zobrazení jedné otázky, ten se nazývá „_poll_question“ (tyto podpohledy patří modelu *PollQuestion*).

Podpohled „_poll_question“ podporuje volitelný parametr – id uživatele, jehož odpověď na otázku (pokud existuje) chceme zobrazit. I když to zde nepotřebujeme, protože chceme zobrazit odpověď aktuálního uživatele, použijeme toto na stránce „odpovědi studenta“, která je popsaná v sekci 4.2.8.

Podpohled „_poll_question“ udělá to, že zobrazí zadání otázky a v závislosti na tom, zda už existuje odpověď konkrétního uživatele na konkrétní otázku, nebo ne, zobrazí podpohled „_new“, nebo „_edit“ (oba z nich už patří modelu *PollAnswer*).

- Podpohled „_new“ představuje vytvářecí formulář s tlačítkem „Odpovědět“. Ten zobrazí možné odpovědi nebo vytvoří prázdné pole, kde uživatel bude moci napsat svou odpověď. Po stisknutí tlačítka se odešle požadavek na vytvoření odpovědi, který zpracuje metoda create (popsaná v sekci 3.3.3).

- Podpohled „_edit“ představuje obyčejný editační formulář, ve kterém jsou vyplněné odpovědi uživatele. Odlišností ale je, že v tomto formuláři nelze nic měnit (pole jsou neaktivní) a není tlačítko, které změny odesílá (to uživatelům nepovolujeme).

Povolení studentům odpovídat na otázky jen během definovaného intervalu času

Splnění tohoto funkčního požadavku zajistíme pomocí upřesnění práva na odeslání odpovědi na otázku ve třídě **PollAnswerPolicy** (třídy tohoto typu byly popsané v sekci 2.3.1).Pokud časový interval pro odpovídání na otázky v anketě byl definován a zároveň se ještě nezačal, nebo už skončil, tak uživateli uložit otázku nepovolíme a zobrazíme hlášku (popsáno v sekci 2.3.1), že tuto akci provést nemůže.

4.2.4 Editace události

Editace události funguje na stejném principu jako vytvoření. Místo *new* a *create* ji ale obsluhují metody *edit* a *update*. Jedinou odlišností je, že nedovolujeme uživatelům měnit typ události, což by znamenalo příliš velkou změnu. Toto rozhodnutí je konzistentní s ostatními částmi MARASTu, kde kupříkladu nelze měnit typ vytvářeného příkladu. Proto nepotřebujeme rozdělovat proces editace na dvě stránky a stačí jen jedna.

Detailem specifickým pro implementaci je to, že pokud před editací atribut *anonymous_responses* byl nastaven na *true* a už byly přidané nějaké anonymní dotazy (pro událost typu Q&A) nebo anonymní odpovědi na otázky (pro událost typu anketa), tak nedovolíme tento atribut měnit. Jinak by to celý model komplikovalo. Toto omezení implementujeme jak v pohledu (pomocí neaktivního pole a slovního popisu), tak i v řadiči (v metodě *update*), aby nějaký zkušenější uživatel nedokázal odeslat na server specifický požadavek, který by tuto změnu provedl.

4.2.5 Nová otázka

Tuto stránku implementujeme podobným způsobem, jako jsme implementovali vytvoření nové události. Také ji rozdělíme na dvě a implementujeme stejným způsobem jak na stráně pohledů, tak i na straně řadiče. Také v závislosti na volbě v prvním kroku vytvoření nové otázky, budeme ve druhém kroku zobrazovat různé atributy pro vyplnění (pro textové otázky jenom zadání, pro *singlechoice* a *multichoice* ještě počet možností a samotné možnosti odpovědí). Při vytvoření, v závislosti na typu události, do atributu *result_json* zapíšeme buď pole s 10 prvky, nebo prázdné asociativní pole.

4.2.6 Editace otázky

Táto stránka je realizovaná podobně jako stránka s editací událostí. Také nepovolujeme měnit typ otázky.

4.2.7 Výsledky otázky, výsledky ankety

Jak bylo zmíněno v sekci 3.2.7 (ve fázi návrhu), chtěli jsme implementovat dvě možnosti a potom si z nich vybrat jednu. Po implementaci ale bylo rozhodnuto obě možnosti v systému nechat, čímž docílíme toho, že bude rozšíření obsahovat výhody obou variant řešení tohoto problému. Jedinou diskutabilní nevýhodou toho, že necháme obě možnosti, je to, že stránka událostí bude obsahovat více tlačítek (pro učitele), tato nevýhoda ale je nepodstatná.

Při implementaci těchto stránek byl použit atribut *result_json* (popsaný v sekci 3.1.2). Ten obsahuje agregaci odpovědí studentů, a proto se velmi pohodlně může použít pro znázornění výsledků. Pro znázornění výsledků bylo využito rozšíření frameworku *Ruby on Rails*, které se nazývá *Chartkick*[14]. Toto rozšíření poskytuje jednoduchý způsob vykreslení diagramů z dat, která předáme. Data předáváme ve formě asociativního pole, kde klíče jsou jednotlivé možnosti odpovědí a hodnoty jsou počty opakování těchto možností.

Rozšíření také poskytuje různé varianty diagramů (koláčový, sloupcový, atd.), proto jeho použitím splníme funkční požadavek (pro události typu Anketa) „Různé varianty zobrazení výsledku“, který je vypsán v sekci 2.5.3.

V tomto rozšíření budeme poskytovat tři typy diagramů: koláčový (*pie chart*) a dva druhy sloupcových (*column chart*, *bar chart*). Jelikož *column chart* a *bar chart* se do češtiny překládají jako *sloupcový graf*, v uživatelském rozhraní necháme anglické názvy.

Pro samotné zprovoznění stránek poskytujících výsledky použijeme parametry předávané v odkazu. Podle nich budeme zobrazovat požadované typy diagramů. Pro stránky zobrazující výsledky jedné otázky předáváme jenom jednu hodnotu. Pro stránky s výsledky všech otázek v rámci události typu

[14] https://chartkick.com/

Anketa, předáváme pole, kde i-tý prvek použijeme pro vykreslení požadovaného typu grafu.

Pro samotné vykreslení diagramů v řadiči byla vytvořena speciální metoda (zmíněná v sekci 3.3.2), která poskytuje samostatný zdroj dat pro grafy. Využití této metody je doporučeno na oficiální stránce rozšíření *Chartkik* kvůli zrychlení procesu vykreslení diagramů.

4.2.8 Odpovědi studenta

Podle návrhu této sekce (uvedeno v 3.2.8) ji rozdělíme na dvě stránky. První z nich bude obsahovat výběr studenta, jehož odpovědi chceme prohlížet; druhá zobrazuje samotné odpovědi vybraného studenta.

Obě tyto stránky jsou implementované samostatnými pohledy. Každá je obsluhovaná speciální metodou v řadiči *EventsConroller*.

Stránku se seznamem studentů zprovozníme pomocí metody select_student (popsaná v sekci 3.3.1). Tuto metodu implementujeme pomocí atributu tabulky PollQuestion – users_participated_json (popsaného v sekci 3.1.2). Metoda udělá to, že projde všechny otázky patřící k události typu Anketa, a slije pole obsahující studenty, kteří na konkrétní otázky odpověděli. Toto výsledné pole předáme do pohledu, kde související studenty zobrazíme (a ke každému přidáme odkaz na jeho odpovědi).

Samotnou stránku s odpověďmi studenta implementujeme pomocí podpohledu „_show_many“, který byl popsán v sekci 4.2.3.2. Proto ale budeme muset přidávat id studenta do „_poll_question“. Podpohled „_poll_question“ také trochu upravíme, aby v případě, že odpověď předaného uživatele na konkrétní otázku neexistuje, nezobrazoval vytvářecí formulář, ale jenom říkal, že tento uživatel ještě na tuto otázku neodpověděl.

4.3 Zbývající požadavky

V této sekci uvedeme popis implementace dvou zbývajících požadavků (celý seznam je uveden v sekci 2.5), o kterých ještě nebylo nic řečeno, ty patří k obecným požadavkům (nejsou vztažené ke konkrétnímu typu události).

Podpora sázecího systému *LaTeX* pro matematické vzorce a značkovacího jazyku *Markdown*

Podpora zmíněných funkčností v MARASTu před začátkem práce už byla. Jelikož jedná se jenom o to, že chceme data speciálním způsobem zobrazovat, stačilo jenom použít speciální příkaz v pohledu. Otázkou bylo, kde přesně to chceme umožňovat. Abychom byli konzistentní na ostatní část MARASTu, pro události povolíme *LaTeX* i *Markdown* jenom u těla události (stejně je to řešeno u příspěvku v blogu). Určitě se *LaTeX* i *Markdown* budou hodit pro zadání otázky v anketě, protože tam učitelé mohou chtít psát víc textu, který

je potřeba strukturovat. Pro možnosti odpovědí už povolíme jenom LaTeX, protože nepředpokládáme, že budou dlouhé.

Komentáře, pomocí kterých jsme implementovali příspěvky v událostech typu Q&A, také *LaTeX* a *Markdown* podporují.

Přívětivost k mobilu, tabletu, atd.

Responzivní design v rámci vytvořených stránek byl zajištěn použitím CSS frameworku Bootstrap, který byl popsán v sekci 2.3.1. Ten nabízí velmi jednoduchý postup vytvoření stránek, které se přizpůsobují obrazovce. Postup spočívá v použití předpřipravených stylů, které definují, jaké rozměry budou mít části stránek v závislosti na rozlišení obrazovky. Rozlišení se dělí na pět typů. Podrobněji si o tom čtenář může přečíst na oficiálních stránkách frameworku. [25].

Podpora anglického jazyka, jakož možného jazyka uživatelského rozhraní

Toto podporuje celý MARAST a bylo by velice nekonzistentní tento požadavek neimplementovat. Pro řešení této úlohy MARAST používá dva soubory, ve kterých se definují překlady slov do českého a anglického jazyka. Tyto překlady potom můžeme využívat v pohledech a hláškách pomocí metody „t“. Například, chceme, v závislosti na tom, českou verzi MARASTu prohlížíme, nebo anglickou, dostat slovo „Předmět“, nebo „Course“. Toto (za předpokladu, že v souborech jsme překlad zadefinovali) uděláme uděláme pomocí volání metody:

```
t('course')
```

Nastavení jazykové verze je v MARASTu realizované pomocí parametru v odkazu. Pokud ten parametr v odkazu není, použije se česká verze.

Kapitola 5

Testování

V této kapitole budou uvedené postupy, které byly použité pro automatické testování implementovaných funkčností. Jak bylo uvedeno v kapitole 2.3.1, pro automatické testování MARAST používá rozšíření *Ruby on Rails*, které se nazývá *RSpec*. Jednou z výhod tohoto rozšíření je to, že ono dovoluje vytvářet různé typy testů. Proto během implementace jsem si využil tři typy testů: testy modelů, testy řadičů a tzv. *feature* testy.

Testy modelů

Slouží, především proto, abychom testovali, že aplikace může detekovat nevalidní stav modelů [1]. Kupříkladu, zde můžeme testovat chování naše aplikace, pokud zkusíme uložit do databáze záznam s chybějícím povinným atributem. Správným chováním by bylo to, že tento požadavek aplikace zamítne a označí záznam jako „nevalidní“. Stejné chování by bylo správným pokud bychom do atributu typu *enum* uložili nepodporovanou hodnotu. Nebo, také, obecně, pokud bychom do atributu s omezeným počtem možných hodnot, uložili hodnotu, která smysl nemá (například, takovým atributem je *number_of_options*, který může obsahovat hodnoty jenom od 1 do 10).

Testy řadičů

Tyto testy jsou rozsáhlejší, než testy modelů. Jejich cílem je ověření správnosti konkrétních metod v řadiči. Jelikož výsledek práce metod v řadiči je závislý na tom, jaký uživatel a v jaký okamžik času metodu spouští, všechny tyto varianty chování máme otestovat. Samotné testy tohoto typu vypadají tak, že zkoušíme spouštět metody a následně ověřujeme, zda odpověď serveru byla očekávaná a správná (kupříkladu, jestli se vykreslila stránka s událostí, nebo naopak, nějaká jiná).

Feature testy

Představují nejpřesnější typ testů. Pomocí nich v aplikaci kontrolujeme nejenom to, že uživatel má přístup ke konkrétní stránce, ale i to, co na ní vidí. Jelikož máme pohledy, které studenti a učitelé používají společně, ale vidí je jinak, pro nás je tento typ testů velmi podstatný. Například, tímto testujeme, že studenti nemají odkaz na vytvoření nové události, když učitelé naopak, mají.

TDD

Poměrně známou testovací technikou používanou v *Ruby on Rails* je *test-driven development* (TDD). Podle této techniky programátor na začátku vytvoří neprocházející testy, a už potom se pustí do implementace, která má splňovat podmínky jednotlivých testů [1].

Tuto techniky jsem si zkusil použit při implementaci, po nějakém času jsem ji ale vynechal kvůli tomu, že pro mě užitečnou nebyla. Problém, který hodně-krát zpomaloval vývoj pomocí této techniky byl v tom, že nemohl jsem odhadnout, jaké části kódu je potřeba testovat v jakém poměru. Před samotnou implementaci vytvářel jsem hodně testů. Po implementaci jsem ale vždy zjišťoval, že vytvořené testy nepokrývají důležité částí kódu. Naopak, se soustřeďuji kolem těch částí, které až tak důkladné testování nepotřebují. Proto v mém případě využití TDD vývoj jenom zpomalovalo.

Závěr

Cílem této práce bylo vytvořit funkční rozšíření existujícího webového portálu MARAST, které by umožňovalo:

- prováděni krátkých anket během výuky.
- pokládání dotazů ze strany studentů během Q&A událostí (nebo také přednášek, cvičení).

Před samotnou implementaci byla provedená analýza existující verze MARASTu, která se zabývala popisem vnitřní struktury portálu, představením používaného frameworku a softwarové architektury. Také nezbytnou části práce byla analýza požadavků potenciálních uživatelů, pomocí které byly upřesněné požadované funkčnosti a jejich priority, které následně byly navržené, implementované a otestované. Jelikož zmíněný portál je velký projekt, který je vyvíjen už dlouhou bodu, pro implementaci byl využit velký počet již hotových funkčností a stačilo vytvořit jenom tři nové modely (v rámci MVC).

Výstupem práce je nová sekce portálu, která po nasazení bude přístupná všem uživatelům MARASTu a ve které jsou dostupné všechny požadované funkčnosti. Cíle práce jsou tím splněné, což zahrnuje většinu vymyšlených učiteli a studenty funkčností.

Možná rozšíření a vylepšení

Na konci bych chtěl uvést možná rozšíření a vylepšení, na které během psání práce nezbyl čas. Ty, většinou mají sloužit pro zlepšení uživatelské přívětivosti.

První z nich je použití AJAX (*Asynchronous JavaScript and XML*) pro odeslání požadavků z pohledů, tím se dá dosáhnout zmenšení počtů celkového překreslení stránek. Což by se hodilo, například, pro stránku s výsledky všech otázek patřící anketě, nebo také obecně pro stánku s anketou, kde uživatelé odesílají odpovědi.

Druhá spočívá v povolení učitelům měnit pořadí otázek v anketě. Implementovat toto bychom mohli, například, pomocí dalšího atributu v tabulce *PollQuestion* a dvou metod v řadiči. Takové řešení v MARASTu funguje pro jiný model.

Bibliografie

1. SLAVÍK, Kryštof. *Systém pro správu písemných testů a zkoušek*. Praha, 2016. Bakalářská práce. ČVUT v Praze, Fakulta informačních technologií, Katedra softwarového inženýrství.
2. Ruby. *Wikipedia* [online] [cit. 2020-05-11]. Dostupné z: `https://cs.wikipedia.org/wiki/Ruby`.
3. Interpretovaný jazyk. *Wikipedia* [online] [cit. 2020-05-11]. Dostupné z: `https://cs.wikipedia.org/wiki/Interpretovan%C3%BD_jazyk`.
4. Skriptovací jazyk. *Wikipedia* [online] [cit. 2020-05-11]. Dostupné z: `https://cs.wikipedia.org/wiki/Skriptovac%C3%AD_jazyk`.
5. Scripting language. *Wikipedia* [online] [cit. 2020-05-11]. Dostupné z: `https://en.wikipedia.org/wiki/Scripting_language`.
6. SYNBIOZ. The story of Ruby on Rails. *Synbioz* [online] [cit. 2020-05-11]. Dostupné z: `https://www.synbioz.com/en/ruby-on-rails-story`.
7. ERZ, Ondřej. *Nástroj pro optimalizaci výkonu Rails aplikací*. Praha, 2015. Bakalářská práce. ČVUT v Praze, Fakulta informačních technologií, Katedra softwarového inženýrství.
8. Dynamický programovací jazyk. *Wikipedia* [online] [cit. 2020-05-11]. Dostupné z: `https://cs.wikipedia.org/wiki/Dynamick%C3%BD_programovac%C3%AD_jazyk`.
9. FLANAGAN, David. *The Ruby programming language*. Beijing Sebastopol, CA: O'Reilly, 2008. ISBN 9780596516178. Dostupné také z: `https://theswissbay.ch/pdf/Gentoomen%20Library/Programming/Ruby/The%20Ruby%20Programming%20Language%20-%20Oreilly.pdf`.
10. Startup. *Wikipedia* [online] [cit. 2020-05-11]. Dostupné z: `https://cs.wikipedia.org/wiki/Startup`.

11. PUTANOBEN, Ben. The 5 Most Popular Programming Languages of 2019. *Stackify* [online] [cit. 2020-05-11]. Dostupné z: https://stackify.com/popular-programming-languages-2018/.

12. Ruby on Rails. *Wikipedia* [online] [cit. 2020-05-12]. Dostupné z: https://cs.wikipedia.org/wiki/Ruby_on_Rails.

13. HARTL, Michael. *Ruby on Rails Tutorial Learn Web Development with Rails, 6th Edition*. Dostupné také z: https://www.learnenough.com/courses/downloads#ruby-on-rails-6th-edition-tutorial.

14. Ruby on Rails. *Wikipedia* [online] [cit. 2020-05-12]. Dostupné z: https://en.wikipedia.org/wiki/Ruby_on_Rails.

15. Convention over configuration. *Wikipedia* [online] [cit. 2020-05-14]. Dostupné z: https://en.wikipedia.org/wiki/Convention_over_configuration.

16. Model-view-controller. *Wikipedia* [online] [cit. 2020-05-12]. Dostupné z: https://cs.wikipedia.org/wiki/Model-view-controller.

17. Model-view-controller. *Wikipedia* [online] [cit. 2020-05-13]. Dostupné z: https://ru.wikipedia.org/wiki/Model-View-Controller.

18. KLOUDA, Karel; KALVODA, Tomáš. O MARASTu [online]. 2018 [cit. 2020-05-03]. Dostupné z: https://marast.fit.cvut.cz/about.

19. NOVÁČEK, Jan. *Analýza chování studentů v systému MARAST*. Praha, 2018. Diplomová práce. ČVUT v Praze, Fakulta informačních technologií, Katedra softwarového inženýrství.

20. Factory Bot (Rails Testing). *Wikipedia* [online] [cit. 2020-05-17]. Dostupné z: https://en.wikipedia.org/wiki/Factory_Bot_(Rails_Testing).

21. Histogram. *Wikipedia* [online] [cit. 2020-05-18]. Dostupné z: https://cs.wikipedia.org/wiki/Histogram.

22. JavaScript Object Notation. *Wikipedia* [online] [cit. 2020-05-25]. Dostupné z: https://cs.wikipedia.org/wiki/JavaScript_Object_Notation.

23. Asociativní pole. *Wikipedia* [online] [cit. 2020-05-25]. Dostupné z: https://cs.wikipedia.org/wiki/Asociativn%C3%AD_pole.

24. Active Record Associations. *Ruby on Rails Guides* [online] [cit. 2020-06-01]. Dostupné z: https://guides.rubyonrails.org/association_basics.html.

25. OTTO, Mark; THORNTON, Jacob. *Bootstrap* [online] [cit. 2020-06-03]. Dostupné z: https://getbootstrap.com/docs/4.0/layout/grid/.

Příloha **A**

Seznam použitých zkratek

HTML Hypertext Markup Language

CSS Cascading Style Sheets

JSON JavaScript Object Notation

TDD Test-driven development

AJAX Asynchronous JavaScript and XML

Příloha B

Obsah přiloženého flash disku

- `readme.txt` ..... stručný popis obsahu flash disku
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF