TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Systém pro správu akreditačních spisů

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Jan Noll**
Vedoucí práce: prof. Ing. Zdeněk Plíva, Ph.D.

Liberec 2020

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Accreditation files management system

Master thesis

Study programme: N2612 – Electrical Engineering and Informatics
Study branch: 1802T007 – Information Technology

Author: **Bc. Jan Noll**
Supervisor: prof. Ing. Zdeněk Plíva, Ph.D.

Liberec 2020

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Systém pro správu akreditačních spisů

Jméno a příjmení: **Bc. Jan Noll**
Osobní číslo: M18000149
Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie
Zadávající katedra: Ústav informačních technologií a elektroniky
Akademický rok: **2019/2020**

Zásady pro vypracování:

1. Prostudujte strukturu akreditačních spisů, doporučení NAU pro jejich přípravu a formáty jednotlivých příloh. Seznamte se s platformou ASP.NET Core, vyberte framework pro frontend a vhodný nástroj pro kontejnerizaci.
2. Navrhněte strukturu systému pro přípravu zejména listů B-III a C-I; zajistěte hierarchický přístup uživatelů k jednotlivým částem dat, zabezpečený přístup, vyhledávání v datech, logování a systém notifikací. Systém musí umožnit opětovné využití jednotlivých listů pro reakreditace i znovupoužití ostatních dat pro nové akreditace. Současně by měl systém umožnit trvalé uložení stavu akreditace tj. její archivaci.
3. Implementujte systém s vhodným přístupovým rozhraním, ověřte funkčnost generování výstupních formulářů na datech vybrané akreditace (BSP i DSP). Součástí předání práce je i řádně komentovaný kód vytvořených dat a programů.

Rozsah grafických prací:	Dle potřeby dokumentace
Rozsah pracovní zprávy:	40-50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] *MŠMT: dokumenty NAU-VŠ* [online]. [cit. 2019-09-18]. Dostupné z: https://www.nauvs.cz/index.php/cs/metodiky

[2] FREEMAN, Adam. *Pro ASP.NET Core MVC 2*. Seventh edition. London: Apress, [2017]. ISBN 978-1-4842-3149-4.

[3] *Microsoft: firemní podpora .NET* [online]. [cit. 2019-09-18]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/

Vedoucí práce:	prof. Ing. Zdeněk Plíva, Ph.D. Ústav informačních technologií a elektroniky
Datum zadání práce:	9. října 2019
Předpokládaný termín odevzdání:	18. května 2020

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci dne 18. října 2019

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

27. května 2020 Bc. Jan Noll

Poděkování

Rád bych poděkoval panu prof. Ing. Zdeňkovi Plívovi, Ph.D., za cenné rady, věcné připomínky a vstřícnost při vedení diplomové práce. Dále bych rád poděkoval Ing. Jiřímu Jeníčkovi, Ph.D., za součinnost při přípravě produkčního prostředí a v neposlední řadě také své rodině, která byla nápomocna zejména při finálních testech implementovaného systému.

Abstrakt

Diplomová práce je zaměřena na návrh a následnou implementaci systému pro správu akreditačních spisů. Požadavky na tento systém vycházejí z provedené analýzy. Především se jedná o definici nového akreditačního spisu, reakreditaci již existujícího studijního programu, fulltextové vyhledávání v datech, archivace stavu akreditačního spisu a systém uživatelských úkolů a notifikací pro zjednodušení komunikace akademických pracovníků. Dále práce popisuje nasazení vytvořeného sytému do produkčního prostředí pro potřeby Technické univerzity v Liberci a v závěru se zaměřuje na techniky testování použité k ověření správné funkčnosti systému.

Klíčová slova: Informační systém, akreditace, C#, .NET, TypeScript, React.js, puppeteer

Abstract

This master thesis is focused on the design and implementation of accreditation files management system. All system requirements are based on the performed analysis. The main goals are creation of new accreditation files, reaccreditation existing study programs, fulltext search, persistation accreditation file in the archive and managing user tasks and notifications. The thesis also describes the deployment process system to the production environment in Technical University of Liberec. At the end, the thesis focuses on techniques of application testing which are used to verify implemented functionalities.

Keywords: Information system, accreditation, C#, .NET, TypeScript, React.js, puppeteer

Obsah

Seznam zkratek

API	Application Programming Interface, programové rozhraní pro komunikaci s aplikacemi
BE	Backend
CORS	Cross-Origin Resource Shared, mechanizmus pro ochranu zdrojů
DOM	Document Object Model, objektově orientovaná reprezentace HTML
FE	Frontend
HTTP	Hypertext Transfer Protocol, protokol pro přenos hypertextových dokumentů
IdP	Identity provider, správce identit systému Shibboleth
JSON	JavaScript Object Notation, způsob zápisu dat
JVM	Java vitual machine, abstraktní virtuální stroj pro interpretaci přeloženého kódu
NAÚ	Národní akreditační úřad pro vysoké školství
ORM	Object-relational mapping, technika pro převoda dat mezi objekty a relační databází
PWA	Progressive web application, progresivní webová aplikace
REST	Representational State Transfer, architektura rozhraní API
SMTP	Simple Mail Transfer Protocol, protokol k přenosu elektronické pošty
SP	Studijní program
SPA	Single page application, webová aplikace renderující obsah na straně klienta
SQL	Structured Query Language, dotazovací jazyk pro komunikaci s relační databází
SSR	Server side rendered, webová aplikace renderovaná na straně serveru
TUL	Technická univerzita v Liberci
UI	User interface, uživatelské rozhraní

Seznam obrázků

1 Úvod a cíle práce

Akreditační spis je nedílnou součástí akreditace každého studijního programu. Tedy i každá vysoká škola je povinna akreditační spisy vytvářet a při akreditaci studijního programu je vždy doložit. Veškeré náležitosti, které musí akreditační spis obsahovat jsou předepsány Národním akreditačním úřadem pro vysoké školství (dále jen NAÚ). Ten všechny tyto pokyny uveřejňuje formou metodických materiálů dostupných elektronicky z [1]. Samotný akreditační spis se skládá celkem z pěti dílčích dokumentů a jeho rozsah řádově nabývá stovek stran. Proto může být jeho ruční sestavení v běžném textovém procesoru obtížné. Obtížnost je navíc ještě umocněna faktem, že některé dokumenty jsou mezi sebou provázány, tedy že informace z listu jednoho dokumentu je promítnuta do několika dalších listů dokumentu druhého. Dalším faktem je, že na sestavení těchto listů pracuje více akademických pracovníků současně.

Cílem této práce je navrhnout a vytvořit informační systém, který zjednoduší přípravu podkladů pro akreditace studijních programů na Technické univerzitě v Liberci a umožní generování dílčích listů spisu ve formátu pdf. Předně se jedná o listy typu B-III a C-I (jmenovitě to je charakteristika studijního předmětu a list personálního zabezpečení), u kterých je kooperace více pracovníků nejkritičtější, a proto se i výsledný dopad realizovaného systému očekává největší.

Systém musí podporovat přípravu podkladů pro akreditaci nových studijních programů, ale i reakreditaci programů stávajících. Současně musí systém umožňovat opětovné použití jednotlivých listů mezi různými akreditacemi, aby bylo možné jednoduše přenést data jednoho předmětu vyučovaného v rámci více studijních programů mezi akreditačními spisy těchto programů. Dále by měl systém, pro usnadnění komunikace v případě vytvoření, či splnění úkolu v rámci sestavení akreditačního spisu, umožňovat odesílání notifikací (i hromadných) mezi akademickými pracovníky. Pro potřeby archivace musí systém umožnit trvalé uložení aktuálního obrazu akreditačního spisu v konkrétním čase ve formátu pdf. Naopak cílem této práce není provedení analýzy akreditačních spisů, ani analýzy stávajících postupů na TUL, a tedy ani provedení návrhu optimalizací těchto postupů. Detailní analýza této problematiky byla zpracována v práci dostupné v [2].

2 Analýza struktury akreditačních spisů, možnosti řešení

Před zahájením návrhu konkrétního informačního systému bude vhodné nejprve zmínit obecně možné postupy řešení vzhledem k požadovaným funkcionalitám a předpokládané struktuře vstupních dat.

2.1 Struktura akreditačních spisů

Jak již bylo zmíněno v úvodu, cílem této práce není detailní analýza akreditačních spisů, jelikož toto téma již bylo dříve detailně zpracováno. Nicméně bude vhodné alespoň krátce popsat obecnou strukturu akreditačních spisů, se kterou bude systém pracovat, což je pro jeho následný návrh nezbytné.

Akreditační spis se skládá z několika různých typů listů. Pro některé typy platí, že jsou obsaženy ve výsledném spisu pouze jednou, některé vícekrát. Typickým příkladem je list popisující charakteristiku předmětu, který je vytvořen pro každý vyučovaný předmět v rámci daného programu. Samotná data každého studijního programu jsou pevně strukturovaná. Tedy je přesně předepsáno, jaká informace, jakého typu je na kterém listu obsažena. Jednotlivé listy pak lze z datového hlediska ve většině případů rozdělit na dvě části. První částí je sekce tzv. metadat, tedy sekce obsahující pevně formátovaná data popisující základní informace o dané entitě. Jako je například název studijního programu, jeho typ, předpokládaná délka studia apod. Druhou částí je sekce jakési charakteristiky této entity. Ta typicky obsahuje uživatelsky formátovatelná data. Konkrétně se jedná například o anotaci předmětu, kde uživatel může formátovat text do odstavců, do seznamů s odrážkami apod. Důsledkem toho je nutnost dynamického přizpůsobení jednotlivých listů jejich obsahu, namísto použití šablon s pevně definovanými rozměry jednotlivých polí, které lze použít v první části. Mimo jiné některé listy obsahují seznamy dynamicky vkládaných položek, což také podporuje předchozí požadavek dynamického přizpůsobení obsahu. Obecně také platí, že obsah (tedy i struktura) některých listů je závislá na typu akreditovaného studijního programu.

Dále pro jednotlivé listy platí, že neobsahují žádné grafické prvky (jak bitmapové, tak ani vektorové). Veškerý obsah je tedy tvořen výhradně daty textového charakteru. Případně je možné do charakteristických polí zapsat URL adresu odkazující na nějaký externí prvek, jak se tomu dělo i v již existujících akreditačních spisech.

2.2 Možná řešení

Informační systém může být řešen, respektive implementován v zásadě dvěma různými způsoby. A to buď jako nativní aplikace, nebo jako webová aplikace.

2.2.1 Nativní aplikace

Nativní aplikace je taková aplikace, která je spustitelná přímo v prostředí uživatelova počítače. Z toho důvodu musí být vyvíjena pro konkrétní platformu. Vzhledem k tomuto faktu je pro implementaci nejsnazší použití některého multiplatformního programovacího jazyka, jakým je například Java, který samotný kód spouští v abstraktním virtuálním stroji (JVM – Java virtual machine) a není tedy přímo závislý na cílové platformě. Případně lze využít další nástroje jako například Electron, které se snaží o tvorbu nativních multiplatformních aplikací. Konkrétně v případě Electronu probíhá vývoj pomocí standardních webových technologií. Následně je takto vytvořená aplikace zabalena společně s odlehčeným webových prohlížečem Chromium do jednoho spustitelného balíčku. Výsledná aplikace je v podstatě lokálně spuštěná webová aplikace „tvářící“ se jako nativní. V tomto případě však nelze plně využít potenciál nativních aplikací, protože vytvořená aplikace nikdy nedosahuje takové míry integrace s hostitelským systémem, jako pro konkrétní platformu vyvíjená nativní aplikace.

Obecně je vhodné využití nativních aplikací v případě, kdy uživatel pracuje pouze nad svými daty a nepotřebuje být spojen s nějakým centralizovaným úložištěm, které by umožňovalo distribuci dat mezi více uživateli. I v těchto případech ale může být použití nativní aplikace výhodné, protože lze implementovat i lokální úložiště dat a s pomocí vhodných synchronizačních mechanizmů umožnit uživateli práci v režimu off-line. Tedy uživatel může v tomto případě pracovat nezávisle bez připojení k centrálnímu datovému úložišti (typicky bez připojení k internetu). Svou práci pak synchronizuje se stavem v centrálním úložišti, odkud je možné ji dále distribuovat ostatním uživatelům. Další výhodou je právě úzká integrace s hostitelským operačním systémem, díky kterému může aplikace využívat jeho vybrané funkce, jako jsou například notifikace.

Nevýhodou nativních aplikací naopak může být nutnost dalších implementačních či post-implementačních úkonů. Mezi ty patří zejména tvorba instalátorů a to jednotlivě pro každou platformu, či například implementace již zmíněných synchronizačních pravidel pro řešení konfliktů vzniklých prací více uživatelů nad nesjednocenou datovou sadou.

2.2.2 Webová aplikace

V případě řešení systému touto formou je implementována jednotná webová aplikace, která je dostupná všem uživatelům. V tom případě je aplikace vystavená pouze na aplikačním serveru, a proto uživatel musí mít pro práci s ním zajištěné spojení (opět zpravidla internetové). Z tohoto důvodu se v dnešní době stále častěji vyvíjí webové aplikace s použitím moderního standardu PWA (tzv. progresivní webové aplikace),

které umožňují částečné použití webových aplikací v režimu off-line. Základní princip je následující, uživatel používá webovou aplikaci standardním způsobem, pokud to uzná za vhodné, přepne tuto aplikaci (samozřejmě pouze v případě, že daná aplikace implementuje standardy PWA) do režimu off-line. V tuto chvíli jsou stažena zdrojová data do uživatelova zařízení a od tohoto okamžiku je možné je procházet i po odpojení od sítě. Aplikace pak musí mít implementované postupy, pomocí kterých tato uložená data průběžně aktualizuje (po opětovném navázání spojení). Zásadním omezením této technologie je přímá závislost na podpoře použitého webového prohlížeče, která v současné době není pro masové požití stále dostačující. Obzvláště adaptace u desktopových prohlížečů je stále pomalá, na rozdíl od těch mobilních, kde se již progresivní aplikace používají namísto nativních mobilních aplikací.

Oproti nativním aplikacím webové aplikace mnohem více usnadňují uvolňování nových verzí. Jelikož, jak již bylo zmíněno výše, aplikace je vystavena pouze na aplikačním serveru. Díky tomu stačí provést aktualizaci právě na tomto jednom zařízení. Ačkoliv se tedy nemusí jednat vždy o jedno zařízení, ale z důvodu rozložení zátěže může být aplikace vystavena na větším poštu serverů, stále je možné provést hromadnou aktualizaci softwaru na všech hostitelských serverech. I tak je tato distribuce snazší oproti nativním systémům, protože jsou všechny servery administrátorovi přístupné, a tak má nad nimi plnou kontrolu. Tato aktualizace zpravidla navodí výpadek poskytované služby, ovšem lze provádět i nasazení nových verzí tzv. bezvýpadkově s použitím více serverů. To ale bývá netriviální úloha z hlediska směrování požadavků, obzvláště, pokud aplikace používá nějaký druh obousměrné komunikace, jako jsou websockety.

3 Návrh informačního systému

Návrh informačního systému se přímo odvíjí od funkcionalit, které jsou od něj vyžadovány. Jak již bylo řečeno v úvodu, detailní analýza problémů a požadavků, ze které vyhází následující návrhy, je popsána v [2].

Samotný systém bude realizován jako webový informační systém, a to zejména z důvodů popsaných v předchozí kapitole. Zejména se jedná o usnadnění práce koncovým uživatelům, tedy oproštění jich od nutnosti instalace specializovaného softwaru na jejich pracovní stanice. Ta by mohla na některé působit nekomfortně. Protože si jednotliví akademičtí pracovníci sami volí operační systém na svých pracovních stanicích, bylo by tedy nutné při vývoji dbát na nutnou multiplatformnost vyvíjených uživatelských rozhraní a rozšířit jejich následné testování. Dále by byl značně ztížen proces budoucích aktualizací a vzhledem k nutnosti práce s jednotnými daty a k faktu, že v aplikaci má být vždy všem dostupný aktuální obraz evidovaných spisů, aplikace by i tak musela pracovat převážně v režimu „on-line“.

3.1 Výběr technologií

Celý informační systém je i přesto navrhován tak, aby byl co nejvíce multiplatformní. Tedy aby volba produkčního běhového prostředí nebyla omezena jedinou platformou, ale aby systém mohl být nasazen jak na serveru s operačním systémem Windows, tak i na některé z distribucí operačního systému Linux. Ačkoliv by se mohlo zdát, že si tato tvrzení protiřečí s výše uvedeným srovnáním s nativními aplikacemi, je nutno podotknout, že v tomto případě nejsou dopady tohoto rozhodnutí natolik zásadní a možnost změnit platformu serveru dodá systému jistou nezávislost.

3.1.1 Volba serverových technologií

Jelikož výše zmíněná multiplatformnost omezuje především výběr serverových technologií, bude vhodné začít právě jejich výběrem. Mezi nejpoužívanější programovací jazyky patří:

- JavaScript
- Python
- Java
- PHP

- C#

To vyplývá ze statistik služby GitHub (dostupných online z [3]), která spravuje přes sto miliónů repositářů obsahujících zdrojové kódy různých aplikací.

Všechny zmíněné jazyky splňují požadovanou multiplatformnost a zároveň nad nimi existují frameworky pro vývoj webových aplikací. Nicméně ne všechny jsou vhodné pro tvorbu komplexnějších systémů, jako je tento navrhovaný. Příkladem může být použití webového frameworku Flask nad jazykem Python, který sami vývojáři prezentují jako „microframework“, jak je uvedeno v online dokumentaci [5]. To přímo neznamená, že by nebyl vhodný pro implementaci rozsáhlejšího systému, ale že obsahuje pouze ty nejzákladnější funkce pro vystavení webového rozhraní. Ostatní (avšak neméně důležité) funkce jako je validace dat HTTP požadavku je nutné doplnit ručně a to buď vlastní implementací nebo použitím rozšiřujícího balíku třetí strany.

Naopak dostatečně silným nástrojem pro vývoj systému tohoto typu by měly být jazyky Java či C#. Tedy pevně typové, kompilované jazyky, které se používají pro vývoj rozsáhlých webových aplikací. Proto bude navrhovaný systém implementován v programovacím jazyku C# na platformně .NET Core. Ta je na rozdíl od starší .NET Framework spouštěna v novém běhovém prostředí, které je již plně multiplatformní. Pro vývoj webových aplikací systému bude použita rozšířená platforma ASP.NET Core. Tj. současně velmi progresivně vyvíjený produkt firmy Microsoft umožňující tvorbu webových rozhraní, který díky nabídce mnoha modulů (jak přímo od firmy Microsoft, tak od třetích stran) může být vhodnou volbou pro řešení komplexních úloh. Výhodou je, že platforma .NET Core nenabízí pouze webové rozhraní, ale umožňuje i tvorbu konzolových aplikací a služeb spouštěných na pozadí. To například s použitím programovacího jazyka PHP není možné (bez použití dalších podpůrných procesů jako je CRON atp.). Díky tomu je možné na jediné platformě provozovat jak samotnou webovou aplikaci, která je pro funkci systému klíčová, tak i koordinovat úlohy na pozadí, jako je periodická kontrola stavu dat, řízené odesílání notifikací, či zajištění jednorázových databázových migrací atp. Použití jediné platformy umožní samozřejmě i sdílení kódu napříč dílčími celky systému, tedy i zrychlí a usnadní vývoj.

Druhou a neméně důležitou částí informačního systému je úložiště dat. Ačkoliv v dnešní době je trendem používat moderní databáze typu NoSQL, jako je například MongoDB či Cassandra, v případě typu dat tohoto systému jejich použití není opodstatněné. Tyto databáze zpravidla ukládají data na principu „key-value“, tedy k unikátnímu klíči uloží libovolnou hodnotu nebo celý objekt (zpravidla ve formátu JSON). Databáze tedy nemá pevně definovanou strukturu dat. Jak již ale bylo zmíněno výše, v navrhovaném systému se jedná většinově o přesně strukturovaná data, proto bude výhodnější použití relační databáze. Vyhovující bude relační databáze PostgreSQL, která splňuje podmínku multiplatformnosti (na rozdíl od Microsoft SQL serveru, který je stále primárně určen pro Windows servery). Současně také, na rozdíl od velmi hojně využívané databáze MySQL, lépe zvládá konkurentní požadavky (jak je uvedeno v [4]). Díky tomu je schopna dosáhnout rychlejších odezev v případě větší zátěže. Výhodou je i oficiální podpora pro platformu .NET Core, pro

kterou jsou publikovány balíčky obsahující připravené konektory pro práci s databází.

3.1.2 Technologie na straně klienta

Klientská aplikace bude také samozřejmě realizována formou webové aplikace. Obecně lze k jejímu vývoji přistupovat dvěma různými způsoby.

Prvním přístupem je frontendová aplikace renderovaná na serveru (zkráceně SSR – Server side rendered). Taková aplikace vrací na požadavek klienta kompletně složenou webovou stránku s veškerým obsahem. Z pohledu uživatele je to stav, kdy při každém přechodu na jinou stránku je vždy stránka znovu celá načtena nehledě na fakt, jestli se daná část mění, nebo ne. Příkladem těchto aplikací jsou aplikace vytvářené pomocí návrhového vzoru MVC, který je mimo jiné defaultně podporovaný platformou ASP.NET Core.

Druhým možným přístupem jsou tzv. SPA aplikace (z anglického Single page application), které se začaly používat až v posledních letech, a to zejména z důvodu masivního rozšíření frameworků usnadňující vývoj těchto aplikací (jako je React.js, Angular atd.). Jedná se o naprosto opačný přístup oproti již zmiňovaným aplikacím typu SSR. Zpravidla jsou v tom případě při prvním otevření stránky stažena všechna zdrojová data celého webu a následně jsou dynamicky překreslovány pouze ty části stránky dle toho, jaké kroky uživatel provádí, a tedy jaké části stránky se mění. V případě, že uživatel přejde například z jednoho článku na druhý, nedochází ke znovu načtení celé stránky, ale pouze se načtou ze serveru data o druhém článku a na stránce se překreslí pouze tato část.

Oba přístupy mají samozřejmě své výhody i nevýhody. Hlavní výhodou SPA aplikací je právě „chytré“ překreslování částí aplikace bez nutnosti načítat vždy celou stránku včetně jejích statických částí. Díky tomu je možné do jisté míry usnadnit vývoj, protože při přechodu z jedné stránky na druhou se neztrácí kontext a jsou přenášena pouze nutná data (na druhou stranu je nutné vhodně kontext vytvářet při přímém otevření konkrétní stránky). Tím lze zpravidla dosáhnout rychlejších odezev aplikace. Zásadní slabinou SPA může být právě nutné načtení celého webu při prvním požadavku. To u větších aplikací vyžaduje jednorázové stažení i jednotek MB, což nemusí být pro uživatele s pomalým připojením komfortní. Tyto situace lze nicméně také do jisté míry optimalizovat pomocí rozdělení zdrojových souborů do více balíčků, které se stahují až v případě potřeby. Je tedy možno například definice administrátorského rozhraní oddělit od zbytku webu a stahovat je až v okamžiku, kdy na něj administrátor přejde. A protože běžní uživatelé tuto část nepotřebují, vůbec ji nestahují. Zásadní problémy SPA se objevují ve chvíli, kdy je třeba vytvořený web tímto standardem optimalizovat pro vyhledávače. Jelikož se data načítají do stránky asynchronně, indexovací roboti zpravidla nezvládají správně stránku načíst, procházet její strukturou apod. Proto se v těchto případech, kdy je třeba tyto optimalizace provádět (zejm. komerční weby, e-shopy), volí vývoj s použitím SSR technologií, či kombinací obou přístupů.

V případě navrhovaného systému bude použit přístup SPA. Jelikož se jedná o interní aplikaci, není potřeba provádět žádné optimalizace pro vyhledávače a zároveň

tím bude docíleno větší dynamičnosti uživatelského rozhraní. Serverová část bude tedy „servírovat“ statickou část webu a aplikační rozhraní, ze kterého bude uživatelské rozhraní načítat data.

3.2 Návrh architektury informačního systému

Následně je nutno rozhodnout o architektonickém rozvržení navrhovaného systému. Tedy rozhodnout, zda systém koncipovat jako jednu monolitickou aplikaci, která implementuje veškeré požadavky plynoucí z analýzy, či systém rozdělit do více samostatných aplikací. Každá jednotlivá aplikace by implementovala pouze část logiky systému jako celku. A případně pak také rozhodnout o rozdělení na konkrétní části.

V případě implementace monolitické aplikace je oproti rozdělenému systému na menší služby jednoznačně výhodou zjednodušený vývoj a následné nasazení. Protože se jedná o jedinou aplikaci, je vývoj zjednodušen o implementaci komunikačního kanálu, pomocí kterého by jednotlivé aplikace mezi sebou komunikovaly. Jelikož není třeba integračních testů, které ověří dostupnost tohoto komunikačního kanálu mezi jednotlivými službami, zjednodušuje se i následné testování. V případě rozdělení logiky systému do více nezávislých služeb, tzv. „mikroslužeb“ naopak získáváme oproti monolitické aplikaci výhodu ve chvíli, kdy je třeba začít škálovat, tedy v případě větší zátěže začít navyšovat výkon. Zpravidla nejsou všechny části systému zatěžovány rovnoměrně, a tak je výhodou možnost škálovat pouze ty aplikace, které jsou aktuálně vytížené a tím i uspořit provozní náklady. Další výhodou může být možnost aktualizace pouze části systému (ať již při opravě chyby nebo při implementaci nové funkce), při které není třeba odstavit celý systém, ale pouze zasažené části. Naopak nevýhodou může být i ztížený proces přípravy běhového prostředí, kdy je třeba oproti monolitu nasadit větší množství služeb a vhodnou konfigurací umožnit jejich komunikaci.

3.2.1 Rozdělení systému do služeb

V případě implementace našeho systému budeme postupovat zcela jednoznačně rozdělením systému na více aplikací/služeb. Nicméně navrhovaný informační systém není tak rozsáhlý, respektive data zpracovávaná v tomto systému jsou jednotná a tedy i špatně separovatelná. Proto budou všechny aplikace využívat jako centrální úložiště všech dat jedinou databázi. Nebude se tedy jednat o pravé mikroslužby ve smyslu, jak bylo nadefinováno výše, jelikož použitím jediné sdílené databáze již jednotlivé služby přestávají být zcela nezávislé. Ačkoliv použitý databázový server PostgreSQL je schopen obsluhovat konkurentní požadavky s minimálním využitím zámků nad daty, i tak tyto situace nejsou zcela vyloučeny.

Největším očekávaným přínosem separace do více služeb je právě výše zmíněná možnost škálování a do jisté míry i zjednodušení a zpřehlednění zdrojových kódů jednotlivých aplikací dosaženým právě logickou separací do menších celků. Výše zmiňované nevýhody způsobené komplikovanou publikací všech služeb na běhové prostředí budou eliminovány vytvořením automatizačních skriptů, které budou s minimální-

mi ručními zásahy automatizovaně provádět nasazení na produkční prostředí a to včetně případných úprav databázové struktury či jiných nutných datových migrací.

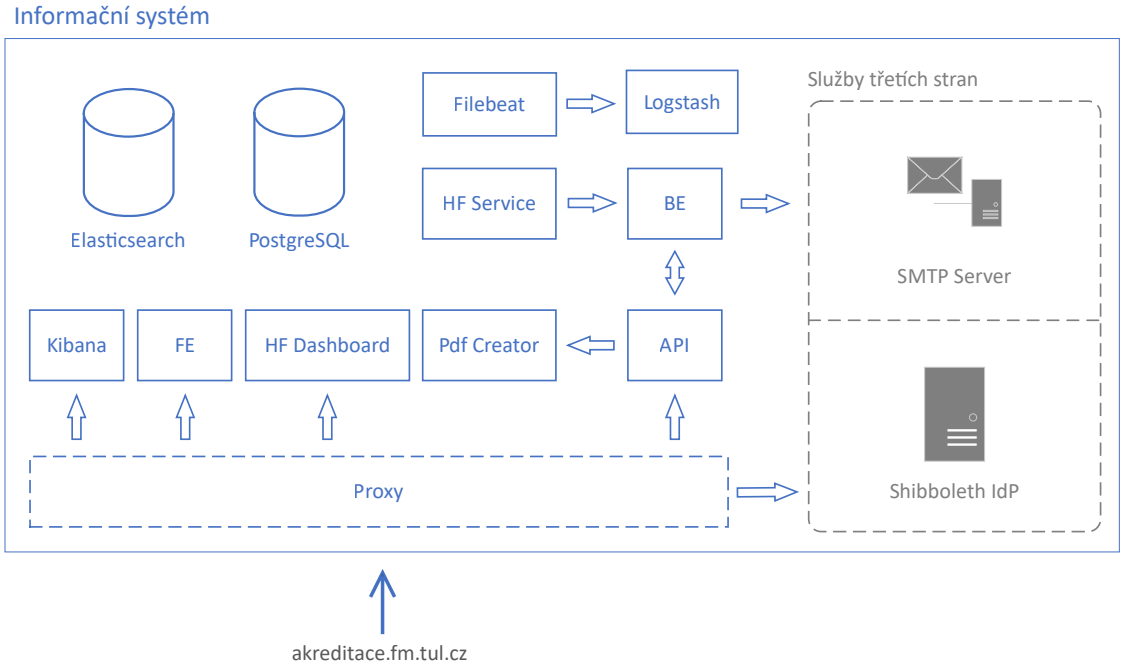
Rozdělení systému do více služeb znesnadní i potenciální hledání chyb. Každá služba totiž vytváří samostatné log soubory v oddělených direktorářích, a tak sledování zaznamenaných akcí uživatele (v těchto souborech), které prostupují přes více služeb může být obtížné. Proto bude výhodné vytvořit mechanizmy pro seskupování těchto záznamů. K tomu bude použit tzv. ELK stack, tj. spojené tři produkty - Elasticsearch, Logstash a Kibana. Přičemž Elasticsearch je samotná databáze, do které jsou ukládány jednotlivé záznamy. Její výhodou je jádro postavené na Apache Lucene, které zvládá pokročilé metody fulltextového vyhledávání. Ty lze využít i pro potřeby fulltextového vyhledávání ostatních (aplikačních) dat systému. Logstash slouží k agregaci vstupních logů a Kibana je již pouhé webové rozhraní pro správu a vizualizaci uložených dat. Tento „podsystém“ lze použít i pro monitoring aplikací/serveru, včetně upozorňujících notifikací. Výslednou architekturu navrhovaného systému lze vidět na obr. 1. Modře jsou označeny části systému, které jsou implementovány v rámci této práce. Šedě pak ostatní externí systémy, které jsou integrovány pomocí dostupného aplikačního rozhraní. Šipky mezi jednotlivými bloky značí směr iniciace komunikace. Jmenovitě se pak jedná o následující aplikace:

- FE
- API
- PdfCreator
- BE
- HF Service
- HF Dashboard
- Kibana
- Logstash
- Filebeat

Jak je dále vidět, veškerá vnější komunikace probíhá přes reverzní proxy, která rozděluje požadavky na jednotlivé aplikace dle URL a disponuje privátním klíčem pro šifrování komunikace. Dále je schopna provádět základní balancování, tj. překládat požadavky na více aplikačních serverů pro optimální rozložení zátěže. Tato funkcionalita bude ve výchozím stavu vypnuta, ale v případě navýšení zátěže v budoucnu je možné snadnou úpravou konfigurace tyto postupy aktivovat.

Aplikace FE

Aplikace FE (frontendu) představuje jednotné uživatelské rozhraní pro celý systém. Aplikace bude implementována formou statických webových stránek a veškerá data bude získávat výhradně z jednotného API.



Obrázek 1: Blokové schéma navrženého informačního systému

Aplikace API

Aplikace API je jedinou aplikací v systému poskytující aplikační rozhraní mimo uzavřený systém. Jedná se tedy o jediný koncový bod pro komunikaci s FE. API dále komunikuje „napřímo“ (tedy bez překladu na reverzní proxy) s aplikací PdfCreator a BE.

Aplikace PdfCreator

Aplikace PdfCreator slouží výhradně ke generování souborů pdf. Jako zdroj pro konverzi jí slouží data ve formátu HTML, která následně aplikace na pozadí renderuje do webové stránky a následně z ní provádí export do pdf. Jedná se zároveň o jedinou aplikaci (vyjma fontendu), která není implementovaná pomocí frameworku .NET Core, ale pomocí Node.JS naprogramovaná v jazyku JavaScript. Důvodem tohoto řešení je nedostupnost dostatečně robustních knihoven pro .NET Core, které by převod z HTML do pdf umožňovaly.

Aplikace BE

Aplikace BE (backendu) slouží ke zpracování úloh na pozadí. Mezi příklady těchto úloh patří zejména periodické hledání podnětů k notifikacím, samotné zpracování a odeslání notifikace, či zpracování administrátorských úkolů, které je potřeba zpracovávat synchronně v přesném pořadí.

Aplikace HF Service

Aplikace HF Service je jednoduchá služba implementující funkce frameworku Hangfire, který slouží ke správě a zpracování úkolů na pozadí. Konkrétně bude použit pro plánování opakujících se událostí, jako je již zmíněné hledání podnětů pro notifikace atp. Zároveň slouží jako implementace front, které budou použity pro také již zmíněnou synchronizaci procesů. Aplikace nebude žádné ze zmíněných činností vykonávat, ale pouze je plánovat. Samotné vykonávání bude provádět BE, který bude službou za tímto účelem kontaktován.

Aplikace HF Dashboard

Aplikace HF Dashboard je určena výhradně pro systémové administrátory a umožňuje monitoring a základní správu aplikace HF Service. Konkrétně umožňuje kontrolu proběhlých úloh, zapnutí či ruční aktivaci opakujících se úloh atd.

Aplikace Kibana

Aplikace Kibana je opět určena výhradně pro systémové administrátory. Kibana je vizualizační nástroj dat uložených v databázi ElasticSearch. Umožňuje vytvářet a ukládat filtrovací pravidla pro selekci uložených dat. Současně podporuje tvorbu přehledových obrazovek s použitím datových a grafických prvků. V případě našeho systému bude použit výhradně systémovými administrátory pro přístup k logům systému a správě indexovaných dat pro fulltextové vyhledávání.

Aplikace Filebeat

Aplikace Filebeat je open-source aplikace pro sběr log souborů. Cílem této aplikace je pouze kontrolovat změny/přírůstky ve všech log souborech a odesílat je do sběrné aplikace Logstash. K jednotlivým datům navíc přiřazuje příznaky, aby je bylo možné v dalších krocích rozlišit.

Aplikace Logstash

Logstash je jediným sběrným místem přicházejících dat do databáze Elasticsearch. Tato aplikace definuje tzv. pipeline, která obsahuje transformační pravidla příchozích log souborů. Tato pravidla se liší dle zdrojové aplikace a obsahují rozdělení vstupního řetězce na jednotlivé parametry. Případně obsahují i konfigurace dalších pluginů, které provádějí další transformace.

Služby třetích stran

Navrhovaný systém bude používat dva různé externí systémy. Prvním je SMTP server, který bude sloužit pro odesílání e-mailových notifikací. Druhým je Shibboleth Identity Provider (zkráceně IdP) nutný pro funkci jednotného přihlašování, používaného na TUL. Ten poskytuje základní informace o přihlášeném uživateli.

3.2.2 Komunikace mezi službami

Ihned po rozvržení systému do jednotlivých služeb je nutné nadefinovat způsob komunikace mezi těmito službami. Implementovat lze komunikaci synchronní, či asynchronní. V případě synchronní komunikace odešle zdrojová služba požadavek na cílovou službu a vyčkává na odpověď. Maximální doba čekání je samozřejmě limitována nastavením tzv. „timeout“ konstanty. V případě asynchronní komunikace zdrojová služba odešle zprávu, ale nečeká dále na odpověď a pokračuje ve vykonávání své činnosti. Mimo jiné asynchronní komunikace musí být použita v situacích, kdy probíhá komunikace s více příjemci, kde z podstaty kontaktování více příjemců nemůže být komunikace synchronní (každý příjemce obecně zpracuje data za různý časový úsek). Zpravidla však v tomto případě služby nekomunikují napřímo, ale využívají dalšího prostředníka, kterým bývá nějaký message broker (například volně dostupný RabbitMQ). Tedy software, který implementuje mezi jednotlivé služby fronty, přes které odeslané zprávy procházejí. Tedy zdrojová služba pouze vloží zprávu do fronty a až přijde na řadu, je předána cílové službě. Výhodou těchto prostředníků také bývá, že podporují funkce směrování, tedy přijmou jednu zprávu a dle předem nadefinovaných pravidel ji uloží do front cílových služeb.

Komunikace v navrhovaném systému bude mít vždy jednoho příjemce, tedy z tohoto důvodu asynchronní komunikace není nutně vyžadována. Dále je naopak například u aplikace PdfCreator synchronní komunikace vyžadována (uživatel musí na žádost dostat vygenerované pdf). Veškerá komunikace mezi jednotlivými službami bude implementována pomocí protokolu HTTP, tedy synchronně. Jak již ale bylo řečeno výše, použitá služba Hangfire implementuje také již zmiňované fronty, kterými docílíme v některých případech vyžadované asynchroničnosti. Jako je odesílání notifikací, které probíhá na pozadí bez nutnosti čekání uživatele na dokončení, které v případě více příjemců může být časově náročnější (řádově i nižší desítky sekund). Konečný postup může být takový, že služba provede synchronní požadavek na BE, ten provede „zafrontování“ požadavku pro Hangfire službu a vrátí zdrojové službě odpověď. Samotný úkol je pak zpracován asynchronně.

3.3 Návrh hostitelského prostředí

Jak již bylo zmíněno výše, systém je navrhován výhradně pro potřeby Technické univerzity v Liberci, proto bude i zde výsledný systém hostován. Při návrhu je tedy třeba dbát předem daných omezení, která univerzita stanovuje.

Pro potřeby navrhovaného systému bude univerzitou poskytnut virtuální server vybrané konfigurace. Jediné zásadní omezení se týká volby operačního systému serveru. Ten musí být z licenčních důvodů založen na linuxovém jádře. Konkrétní distribuce nejsou již dále omezeny, ale pouze doporučeny. Z důvodu zjednodušení počáteční konfigurace serveru (či případné migrace v budoucnosti) bude vhodné jednotlivé aplikace systému umístit do tzv. kontejnerů, tj. zavedení dalšího stupně virtualizace mezi aplikaci a virtuální server. Výhodou oproti klasické virtualizaci je především v úspoře výpočetního výkonu, protože virtualizované kontejnery využívají jádro hostitelského operačního systému a není tedy nutné vynakládat prostředky

pro obsluhu dílčích operačních systémů. Další výhodou je možnost definice automatizovaných skriptů sloužících k vytvoření obrazu kontejneru, který se následně spouští.

Mezi nejpoužívanější kontejnerizační nástroje se řadí jednoznačně nástroj Docker, jehož výkonné jádro je vyvíjeno jako open source. Jako další alternativy lze jmenovat například rkt, které se snaží o co největší bezpečnost, nebo například LXD, které je z ostatních zmíněných nejspíše nejrobustnější. Oproti ostatním disponuje LXD funkcemi, jako je pokročilá správa virtuálních sítí, či pokročilá správa datového úložiště (umožňující rozdělení úložišť, omezení velikostí atp.).

Pro navrhovaný systém však tyto pokročilé funkce nejsou nutné, proto bude výhodnější použít „odlehčenější“ alternativu. V tomto případě bude optimální použití nástroje Docker a to zejména z důvodu dostupnosti vysokého množství předpřipravených obrazů, které je možné převzít a snadnou modifikací použít v našem systému. Tyto obrazy jsou distribuovány přes veřejné centralizované, či soukromé registry. Oficiální repositář obsahuje dle dat dostupných z [6] téměř tři a půl milionu různých obrazů.

3.4 Návrh struktury databáze

Po návrhu základní architektury systému je možné přejít ke konkrétnímu návrhu struktury vybrané relační databáze. Grafickou podobu výsledné struktury databáze lze vidět v příloze A.

3.4.1 Popis jednotlivých entit

Následuje základních popis všech implementovaných entit v relační databázi PostgreSQL. Pro všechny uváděné entity dále platí, že implementují vlastnosti pro ukládání tzv. auditních logů. Jedná o slupce obsahující čas vytvoření, jméno autora a čas poslední úpravy a jméno autora této úpravy.

Tabulka *Addresses*

Addresses je obecná tabulka pro ukládání adres v jednotném formátu - ulice, poštovní směrovací číslo, město. Pro jednotlivé záznamy platí, že nejsou v systému samostatně vytvořitelné (ani editovatelné), ale váží se vždy k další entitě (typicky škole).

Tabulka *Programs*

V tabulce *Programs* budou ukládána všechna data charakterizující každý studijní program. Část uživatelsky formátovatelných dat bude před uložením transformována do standardu HTML, ve kterém bude ukládána. V tomto formátu bude následně i používaná pro zobrazení dat na stránce, či pro generování do pdf souboru.

Tabulka *ProgramSpecializations*

Tabulka *ProgramSpecializations* budou sloužit pro ukládání jednotlivých specializací studijního programu. Především se jedná o název specializace a další uživatelsky formátovatelná pole. Pro ty platí převod do HTML stejně jako u programů.

Tabulka *ProgramInvolvements*

ProgramInvolvements je tabulka pro ukládání informací o dalším zapojení osob do uskutečňování studijního programu. Jedná se o zapojení školitele a člena oborové rady. Tato data budou ukládána pouze pro doktroské studijní programy.

Tabulka *Subjects*

V tabulce *Subjects* budou ukládány informace o všech předmětech všech studijních programů. Popis předmětu jakož to uživatelsky formátovatelný vstup bude opět ukládán ve standardu HTML. Další dynamické parametry jako jsou témata přednášek a cvičení či položky studijní literatury, budou serializovány do formátu JSON a jako prostý text ukládány do jednotlivých sloupců tabulky. Tento přístup je možný, jelikož data neobsahují žádné cizí klíče. Nemůže tedy dojít k narušení referenční integrity. Současně nepředpokládáme vyhledávání dle těchto dat a současně nepředpokládáme selekci jednotlivých záznamů. Pro všechny výstupy budou použita vždy všechna tato dynamická data.

Tabulka *SubjectSpecializations*

Tabulka *SubjectSpecializations* bude propojovat předměty se specializací studijního programu. Pro studijní programy bez specializací nebude tedy tato tabulka obsahovat žádná data. Vyjma vazebních klíčů zde bude pro každý záznam ukládám typ předmětu. To je jediný proměnlivý parametr předmětu napříč specializacemi studijního programu.

Tabulka *SubjectGarants*

Tabulka *SubjectGarants* bude sloužit k propojení osoby vykonávající funkci garanta předmětu s daným předmětem. Dále bude obsahovat procentuální zapojení dané osoby do této funkce.

Tabulka *SubjectTeachers*

Analogicky ke garantům bude tabulka *SubjectTeachers* sloužit k propojení vyučujících osob s daným předmětem. Stejně tak bude uloženo jejich procentuální zapojení do výuky. To bude navíc rozděleno na zapojení do přednášek, cvičení a seminářů.

Tabulka *ProgramArchives*

ProgramArchives bude obsahovat informace o archivovaných programech, jako je čas archivace, poznámka autora či cesta do úložiště k archivovanému souboru.

Tabulka *Schools*

Zde budou uloženy školy vedené v systému. Ukládány budou pouze nejzákladnější informace nutné pro sestavení kompletní výstupů. Konkrétně se jedná o název a adresu vysoké školy.

Tabulka *Faculties*

V tabulce *Faculties* budou uloženy fakulty. Pro fakulty je nutné ukládat pouze jejich název a zařazení k vysoké škole. Zavedení fakult je však nezbytné pro přiřazení studijních programu a přiřazení domovských fakult vyučujícím, pro správné řízení opravnění v systému.

Tabulka *Users*

Do tabulky *Users* budou ukládány informace o všech osobách působících v akreditačních spisech. Jedná se tedy jak o aktivní uživatele systému, kteří sami systém používají, tak i o profily externích pracovníků bez přístupu do univerzitních systémů, vytvořených administrátorem. Takové ručně vytvořené profily neobsahují pouze některá klíčová data (jako uživatelské role) používaná pro autentizaci uživatele na vystavených rozhraních.

Tabulka *UserRelationToFaculties*

Tabulka *UserRelationToFaculties* bude obsahovat seznam pracovních úvazků osob evidovaných v systému.

Tabulka *UserTasks*

V tabulce *UserTasks* budou evidovány všechny uživatelské úkoly. Mezi ukládané parametry patří zejména typ úkolu, stav úkolu, identifikátor entity, níž se úkol týká či autor úkolu (pokud se nejedná o úkol vytvořený systémem).

Tabulka *UserTaskGroups*

Do *UserTaskGroups* tabulky se budou ukládat informace o hromadných úkolech. Tedy o úkolech, které mají více řešitelů. Cílem této entity je „propojit“ všechny vytvořené dílčí úkoly pro jednotlivé pracovníky a umožnit tak kontrolu nad úkolem jako celkem.

Tabulka *TaskQueue*

Tabulka *TaskQueue* bude sloužit pro dočasné uložení informací o úloze uložené ve frontě pro zpracování na pozadí. Ukládána jsou data specifická pro každý typ úkolu a další telemeterická data, která jsou vyplňována v případě neúspěšného zpracování úlohy. Tato telemetrická data jsou konkrétně chybová hlášení a čas posledního neúspěšného pokusu o zpracování.

Tabulka *NotificationQueue*

Tabulka *NotificationQueue* má obdobnou funkci jako předchozí tabulka *TaskQueue*. Slouží k dočasnému uložení dat o notifikaci uložené ve frontě pro její zpracování. Telemetrické údaje jsou ukládány jako v předchozím případě. Dále jsou ukládány informace nutné pro odeslání samotné notifikace (typ zprávy, tělo zprávy, předmět, příjemci). Případně je ukládána vazba na uživatelský úkol, o kterém notifikace informuje.

4 Implementace

Po dokončení obecného návrhu systému lze začít s konkrétní implementací jednotlivých částí systému. Proto jsou v následujících kapitolách nejprve popsány konkrétní použité technologie třetích stran rozdělených dle typu aplikací. Dále následují popisy implementací dílčích funkcionalit systému, jako je fulltextové vyhledávání, autentizace, centralizovaný sběr logů atd. V závěru kapitoly je pak vysvětlen proces nasazení aplikace do produkčního prostředí.

4.1 Použité technologie

Jelikož některé dílčí úkony systému jsou obecné a často se opakující mezi širším spektrem vývojářů, existují balíčky třetích stran řešící dané problematiky. Použití těchto balíčků zefektivní vývoj a tím zajistí čas na implementaci specifických funkcí navrženého systému.

Proto jsou v následujících podkapitolách popsány použité balíčky třetích stran použité při implementaci systému. Ty jsou rozděleny do třech základních sekcí. A to na Frontend, který představuje jednotné uživatelské rozhraní implementované jako single page webová aplikace. Dále na Backend .NET Core představující všechny serverové aplikace implementované s použitím této technologie a na Backend JS, tedy serverovou aplikaci implementovanou pomocí programovacího jazyku JavaScript.

4.1.1 Frontend

Frontendová aplikace představuje, jak již bylo zmíněno, uživatelské rozhraní dodávané formou webové aplikace. Jelikož se jedná o aplikaci, která není renderovaná na serveru, ale v prohlížeči uživatele, aplikace je vyvíjena primárně v programovacím jazyku JavaScript. Proto je pro správu použitých balíčků třetích stran použit správce závislostí *npm*. Ten současně obsahuje i nejrozsáhlejší registr těchto balíčků. Jelikož do tohoto registru přispívá téměř každý open source vývojář JavaScriptových knihoven, lze s jeho použitím snadno použít velké množství připravených balíčků.

Z důvodu zajištění co největší kompatibility s běžně užívanými webovými prohlížeči bude před vystavením aplikace na server probíhat kompilace zdrojových kódů, během které bude kód transformován do standardu, u kterého lze předpokládat vyšší míru podpory moderními webovými prohlížeči. Pro tuto transformaci budou také použity balíčky třetích stran.

React

React.js je open source framework vyvíjený firmou Facebook pro tvorbu interaktivních uživatelských rozhraní. V současné době se jedná o jednu z nejpoužívanějších knihoven. Využívá konceptu komponent, které se renderují do virtuálního DOMu. Pro každou komponentu platí, že má svůj vnitřní stav, který si sama obsluhuje (definuje jeho strukturu) a může dle něj například podmiňovat obsah, který renderuje. Komponenty je také možné při jejich inicializaci parametrizovat. Mezi hlavní přednosti Reactu (a důvodu, proč byl tak rychle rozšířen) patří právě „chytré“ překreslování obsahu, kdy framework při změně vstupních dat překreslí pouze tu část stránky, které se změna dat dotkne.

React Router

Ačkoliv aplikace je implementovaná jako single page, není vhodné vystavit aplikaci na jedinou adresu URL. Uživatelé by přišli o možnost ukládat a sdílet odkazy na jednotlivé stránky. Z toho důvodu je výhodné použít balíček React router, což je rozšiřující knihovna frameworku React, umožňující definici URL adres a přiřazení k nim jednotlivých stránek. A to se zachováním SPA konceptu, kdy při přechodu z jedné stránky na druhou, dochází ke změně URL v adresním řádku, ale nedochází k přenačtení všech statických prvků stránky.

Redux

Redux je knihovna umožňující ukládat stavy sdíleně napříč komponentami. To je pro větší aplikace výhodné, protože není třeba předávat skrz parametry komponent callback metody pro změnu stavu nadřazených komponent. Redux implementuje jednotlivé úložiště, jejichž data si každá komponenta, pokud chce, může injektovat skrz své parametry. Data tohoto sdíleného stavu se chovají jako běžné parametry komponenty. Ta je tedy nemůže sama měnit. K tomu slouží tzv. akce, tj předdefinované metody, které se do komponenty injektují stejným způsobem a slouží k nastavování těchto sdíleních stavů. Použití sdílených stavů samozřejmě není výhodné v každé situaci. Zejména u jednoduchých vizuálních komponent, u kterých se nepředpokládá nutnost ovlivňování jejich dat vně této komponenty, není žádný důvod pro použití sdílených stavů.

Antd

Antd představuje knihovnu pro tvorbu interaktivních uživatelských rozhraní. Použitím této knihovny získáme předem připravené komponenty použitelné v naší aplikaci. Konkrétně framework nabízí prvky pro tvorbu základních layoutů webové stránky, připravená modální okna, nebo například komponentu pro vizualizaci načítání stránky. Největším přínosem tohoto frameworku pro implementovanou aplikaci jsou formuláře. Ty nabízejí všechny potřebné vstupní prvky, jako jsou textová pole, zaškrtávací pole, pole pro výběr datumu, času, tlačítka a další. Současně tyto

formuláře disponují robustním systémem validací vstupních polí. Díky nim lze snadno používat předpřipravené validace (jako je povinné pole, formát e-mailu apod.), i nadefinovat vlastní specifická pravidla.

LESS

Ačkoliv je použit velmi rozsáhlý framework antd, nelze jeho připravené komponenty implementovat na všechny prvky aplikace. Proto je nutno provést „ostylování“ vlastních komponent, či upravit ty existující tak, aby odpovídaly návrhu. Z toho důvodu je výhodné použít LESS, preproces CSS. LESS je jedním z dostupných jazyků, které rozšiřují schopnosti klasického stylopisu CSS, který je prohlížeči běžně podporován. Mezi hlavní přínosy patří možnost definice proměnných (do nich lze ukládat RGB kódy barev, vzdálenosti odsazení prvků apod.), podpora výpočetních funkcí (například pro výpočet tmavšího odstínu barvy atd.), či možnost vnořování samotných stylů do jakési stromové struktury, která usnadňuje orientaci v kódu.

SignalR

Aplikace musí podporovat příjem zpráv ze serveru. Jedním z možných řešení je použití websocketů. Websockety definují komunikační protokol využívající TCP spojení k zajištění plně duplexní komunikace. Knihovna SignalR je nadstavba nad tímto komunikačním protokolem, jejíž použití usnadní implementaci tohoto typu komunikace. Hlavním přínosem je existence serverové verze této knihovny pro platformu .NET Core, díky které lze snadno vytvořit obousměrný komunikační kanál mezi klientskou a serverovou aplikací.

Kompilace kódu

Jak již bylo zmíněno výše, kód bude pro zajištění maximální podpory ze strany prohlížečů před nasazením na produkční prostředí kompilovaný. To ale není jediný důvod. Zdrojový kód nebude programován v jazyku JavaScript, ale v jeho rozšířené verzi TypeScript, která není webovými prohlížeči podporována. Ta oproti JavaScriptu umožňuje práci s datovými typy. To přináší výhody v lepší čitelnosti kódu a díky pevné definici objektů umožňuje snadnější kooperaci více vývojářů, která by bez těchto definic byla velmi obtížná.

Pro provedení všech kompilací a složení produkčního buildu bude použit nástroj webpack. S jeho pomocí je možno s využitím dalších pluginů nadefinovat celý postup složení výstupních balíků s ohledem na typy zdrojových souborů. Tedy zvlášť se definují zdrojové soubory psané v jazyku Typescript, na které se aplikují tzv. „polyfills“. Ty se používají jako rozšíření funkcí některých webových prohlížečů. Obecně lze říct, že pokud některý webový prohlížeč danou funkci nepodporuje, použije se tato ručně implementovaná funkce. Zvlášť se také definují zdrojová data v jazyku LESS. Ty se transformují do stylopisů CSS a opět s pomocí dalších pluginů se provádí další ex post úpravy kódu, jako jsou opravy známých bugů, doplnění prefixů nutných pro správnou funkci ve všech webových prohlížečích apod. Všechny výstupní kódy se minifikují a jejich názvy se doplňují o hash, aby se zajistilo stažení

správného souboru (po případné aktualizaci) v případě, že uživatel ukládá statický obsah do cache. Dále se také definují pravidla pro ostatní (zpravidla multimediální soubory), které se už nijak netransformují, ale pouze přesouvají do výstupní složky a přejmenovávají, stejně jako soubory se zdrojovými kódy.

4.1.2 Backend .NET Core

Backend na platformě .NET Core představuje největší podíl počtu aplikací tohoto typu v systému. Jednotlivé aplikace mezi sebou budou sdílet části kódu. Z toho důvodu jsou vytvořeny samostatné knihovny, které vždy danou logickou část implementují. Na platformě .NET probíhá toto sdílení formou zkompilovaných kódů do knihoven DLL. Nad těmito knihovnami je postaven správce balíčků NuGet. Ten umožňuje vystavit zkompilovaný kód do repositáře, odkud je dále distribuován do jednotlivých aplikací. Každý publikovaný kód vždy nese informaci o číslu verze, podporovaných cílových platformách, dalších závislostech apod. Tyto repositáře mohou být soukromé, či veřejné. Microsoft provozuje jeden veřejný repositář obsahující k dnešnímu dni téměř dvě stě tisíc unikátních balíčků (dostupných na [7]). Lze v něm nalézt téměř všechny open source knihovny pro platformu .NET, či .NET Core, publikované jejich oficiálními autory. V případě implementace navrženého systému budou balíčky třetích stran získány výhradně z tohoto oficiálního repositáře.

Entity Framework Core

Entity Framework Core je nástroj pro objektově relační mapování. Standardně pracuje s relační databází Microsoft SQL Server, ale s použitím vhodných rozšíření je možno ho použít i s relační databází PostgreSQL. Tento nástroj umožňuje oba přístupy k návrhu databáze. Lze tedy postupovat jak přístupem Code-first, kdy jsou nejprve nadefinovány v kódu objekty představující jednotlivé entity a následně z těchto definic automatizovaně vytvořeny tzv. migrace, jejichž aplikací je upravena struktura použité databáze. Případně lze použít přístup Database-first, kdy se nejprve připraví struktura databáze a z ní se automatizovaně vygenerují objekty, do kterých probíhá následné mapování.

V případě tohoto systému je použit přístup Code-first, který umožní snadnou definici databázové struktury přímo v kódu aplikace. Případné změny ve struktuře databáze budou vždy aplikovány pomocí automatizačních skriptů při nasazení systému na konkrétní běhové prostředí. Díky tomu bude vždy zajištěno sjednocení skutečné struktury databáze a nadefinovaných objektů v kódů, které je pro správnou funkci nezbytné. Ačkoliv použití ORM je v mnoha případech efektivní a zrychluje vývoj, v některých specifických případech není jeho použití vždy optimální (vygenerované dotazy stahují v daný okamžik nepotřebná data). Proto je v některých případech využit jen databázový konektor (implementovaný v Entity Frameworku) pro obsluhu spojení s databází, ale není využito objektově relačního mapování. Namísto toho je použit vlastní předem optimalizovaný databázový dotaz.

NEST

NEST je oficiální klient databáze ElasticSearch určený pro platformu .NET. Knihovna implementuje vlastní HttpClient, pomocí kterého komunikuje s rozhraním databáze. Elastic nabízí i druhou knihovnu Elasticsearch.NET umožňující pracovat s databází na nižší úrovni. NEST ale pro navžený systém nabízí dostatečné množství funkcí.

Hangfire

Knihovna Hangfire implementuje mechanizmy pro zpracování úloh na pozadí. V základní neplacené verzi tato knihovna umožňuje zaregistrování úlohy ke zpracování s použitím front (včetně možnosti nastavení priorit), nastavení odloženého zpracování, automatické zpracování opakujících se úloh či zřetězení více úloh za sebou. Dále nabízí předpřipravené jednoduché uživatelské rozhraní pro základní správu a monitoring úloh.

V ročně placených verzích jsou navíc podporovány funkce dávek a další pokročilá nastavení, kterými lze konfigurovat například maximální počet současně spuštěných úloh konkrétního typu apod. Pro použití v tomto systému však budou základní funkce dostačující.

SignalR

SignalR je, jak již bylo zmíněno v sekci Frontend, pomocná knihovna postavená nad technologií websocketů umožňující plně duplexní komunikaci mezi backendem a frontendem. Tato technologie vyžaduje na serverové straně vytvoření tzv. Hubů. To jsou jakési definice všech podporovaných metod/funkcí pro komunikaci tímto kanálem. K těmto metodám se již následně připojují klienti, kteří zde mohou naslouchat příchozím zprávám, či produkovat zprávy vlastní.

NLog

Knihovna NLog umožňuje ukládání pevně strukturovaných aplikačních logů různých úrovní. Data umožňuje ukládat například do souborů, do databáze, či je pouze vypisovat do konzole. Podporuje i automatické mazání starších výstupních souborů.

4.1.3 Backend JS

Backendová aplikace vyvíjená v JavaScriptu je pouze jedna a má pouze jedinou funkci. Nevyužívá žádnou databázi, ani další prostředky. Proto oproti předchozím aplikacím využívá pouze tři balíčky, které jsou také (jako aplikace frontendu) spravovány manažerem *npm*.

Node.js

Node.js je implementace javascriptového jádra V8 z Chrome, rozšířená o další funkce umožňující přístup k souborovému systému, síťovým periferiím apod. Node.js je

asynchronní a událostmi řízený. Jelikož pracuje pouze v jednom vlákně, nevznikají dead-locky. Díky tomu lze snadno vytvářet škálovatelné aplikace.

Puppeteer

Knihovna Puppeteer poskytuje API pro ovládání webového prohlížeče Chrome (či odlehčenějšího Chromium) spuštěného na pozadí, tj. bez grafického uživatelského prostředí. Toho se často využívá pro automatizované testování uživatelských rozhraní, měření výkonosti webových stránek, generování screenshotů apod. V našem systému využijeme funkce knihovny pouze pro generování výstupů pdf.

Kompilace kódu

Jelikož výsledný kód je spouštěný na serveru, u kterého máme zajištěnou podporu aktuálních standardů ES6, není nutně třeba provádět optimalizační kompilace, jako tomu je u Frontendu. Avšak ze stejných důvodů jako tomu bylo u Frontendu probíhá vývoj v typovaném jazyce TypeScript. Pro kompilaci ale nejsou použity žádné další nástroje, jako již zmiňovaný webpack, ale pouze základní Typescript kompiler *tsc*.

4.2 Implementace uživatelského rozhraní

Jak již vyplývá z popisu použitých technologií popisovaných v předchozí kapitole, hlavním konceptem pro vývoj uživatelského rozhraní bylo použití komponent a sdílených stavů. Veškeré stránky jsou proto tvořeny jako samostatné komponenty, které se vykreslují do jednotného layoutu, tvořeného taktéž z komponent. Pro tvorbu opakujících se elementů (jako jsou ne/stránkovatelné seznamy a detaily) jsou navíc vytvořeny bázové komponenty a k nim bázové třídy typu store. S využitím dědičnosti tak je možné snadno vytvořit například všechny stránky se seznamy. A to včetně naplnění daty ze serveru, které byly ukládány do sdílených stavů s pomocí implementací v bázové třídě typu store. Tyto třídy v sobě implementují i všechny základní CRUD (create, read, update, delete) operace, díky kterým je možné snadno vytvářet požadavky těchto typů.

4.3 Implementace aplikačního rozhraní

Rozhraní je implementováno dle standardu REST s využitím controllerů. Vytvořením tzv. akcí v těchto controllerech se definují konkrétní endpointy, které bude aplikace vystavovat. Ty byly připraveny tak, aby jejich výstupy byly co nejvíce přizpůsobeny pro potřeby uživatelského rozhraní. Není tedy vystaveno žádné obecné aplikační rozhraní umožňující dalšímu systému práci s jednotlivými entitami. Naopak je nakonfigurována ochrana CORS (Cross-Origin Resource Shared), která tomuto zamezuje.

Mimo rozhraní typu REST jsou dále vystaveny endpointy implementující technologii websocketů, umožňující odesílání a přijímání push notifikací. Posledním typem

jsou endpointy pro získávání souborů (generování dokumentů a stahování archivovaných programů). Ty nejsou implementovány v controlleru, ale pouze jako tzv. middleware. Ten oproti controlleru neumí jednoduše parsovat a validovat vstupní data, ale naopak umožňuje snadno pracovat na nižší úrovni s HTTP požadavkem a jeho odpovědí. Díky tomu lze například snadno streamovat větší datové soubory, které nechceme před odesláním celé ukládat do operační paměti.

4.4 Implementace fulltextového vyhledávání

Fulltextové vyhledávání je třeba rozdělit do dvou částí. První částí je nadefinování indexů a jejich tvorba, druhou pak jejich samotné prohledávání.

4.4.1 Indexace dat

Protože je pro fulltextové vyhledávání použita databáze ElasticSearch, ale všechna data systému jsou primárně ukládána do relační databáze PostgreSQL, je třeba zajistit, aby data potřebná pro vyhledávání byla ukládána i do databáze ElasticSearch.

Vyhledávání cílí primárně na nalezení studijních programů a předmětů. Proto jsou vytvořeny dva tzv. indexy, tj. oddělené „prostory“, do kterých se data ukládají a následně prohledávají. Jednotlivé entity nejsou ukládány, neboli indexovány, v kompletní podobě, ale pouze jejich části, dle kterých bude probíhat vyhledávání. Ukládány jsou tedy převážně textové řetězce a identifikátory nezbytné pro další procházení v uživatelském rozhraní. V případě studijních programů se jedná o:

- Identifikátor programu
- Název programu
- Jméno garanta
- Název fakulty

V případě předmětů se jedná o:

- Identifikátor předmětu
- Název předmětu
- Identifikátor programu
- Název programu
- Jména všech garantů
- Jména všech vyučujících

Samotné indexování je vždy podníceno změnou dané entity (tj. vytvoření nové, úprava, či smazání existující). Dále bude probíhat pravidelné přeindexovaní všech entit jednou denně, vždy v pravidelnou noční hodinu. To zahrnuje smazání všech stávajících dat a vytvoření nových indexů.

4.4.2 Vyhledávání v datech

Vyhledávání probíhá odděleně nad oběma výše popsanými indexy. Vstupem pro vyhledávání ale není pouze uživatelem zadaný řetězec. Výsledky hledání musí být omezeny pouze na data, která jsou dle pravidel řízení oprávnění konkrétnímu uživateli povolena prohlížet. Proto jsou vyhledávací dotazy rozšířeny o pole povolených identifikátorů programů.

Vyhledávací dotaz je sestaven tak, že hledaný řetězec rozdělí na jednotlivá slova a vyhledává je odděleně. Současně jsou ale podporovány další funkce umožňující bližší specifikaci hledaného výrazu. Podporované jsou uvozovky, kterými je možné označit hledaný řetězec za výraz a hledat jeho celé znění současně. Dále je podporován znak „-“ (pomlčky), jehož uvedením na začátku slova zajistíme omezení výsledků hledání na záznamy neobsahující dané slovo. Posledními podporovanými rozšířeními jsou operátory AND a OR, které se chovají jako běžné binární operátory.

4.5 Implementace generátoru pdf

Jak již bylo zmiňováno výše, úloha generování výstupů pdf je implementována na platformě Node.JS s použitím programovacího jazyku TypeScript. Samotná služba vystavuje rozhraní, kde očekává zdrojová data ve formátu HTML. Po přijetí požadavku, pomocí knihovny Puppeteer, spustí na pozadí instanci aplikace Chromium. V této instanci otevře nové okno a jako obsah nastaví zdrojová data přijatá v požadavku. Následně spustí v instanci proces generování pdf. Pro tuto akci použije předem pevně definované parametry určující velikost výstupní stránky, odsazení od okrajů a zahrnutí pozadí stránky do exportu. Po dokončení generování je vytvořená instance Chromia ukončena a vygenerovaný výstup je ve formátu pole bytů odeslán v odpovědi na požadavek.

S ohledem na fakt, že jsou na pozadí spouštěny další aplikace, je nutné připravit běhové prostředí. Aplikace je spouštěna jako všechny ostatní ve svém vlastím Docker kontejneru. Ten musí disponovat kromě běhového prostředí Node.js pro správnou funkčnost webového serveru, také dalšími balíčky. Proto byl předem připraven nový Docker obraz, který všemi potřebnými balíčky disponuje. Ten vychází z obrazu s běhovým prostředím Node.js a navíc je rozšířen o Google Chrome, jím vyžadované fonty a správce procesů dumb-init. Ten je použit jako vstupní bod kontejneru.

4.6 Implementace autentizace

Na půdě Technické univerzity v Liberci je provozován soukromý poskytovatel identit Shibboleth IdP (zkráceně z anglického Identity Provider). Ten je napojený na adresářový server LDAP a sám poskytuje pouze funkci SSO (Single Sign-On). Tedy funkci centralizovaného přihlašování, díky které uživatel po přihlášení na jednom místě je autentizován na všech službách registrovaných v tomto ekosystému.

4.6.1 Integrace Shibboleth Service Provider

Pro integraci systému do ekosystému Shibboleth je použit Shibboleth Service Provider, nebo-li poskytovatel služby, v aktuální verzi 3. To je dle sdružení CESNET [8] doporučovaná technologie pro integraci s poskytovatelem identit. Tuto technologii lze integrovat s webovými servery IIS a Apache, pro které jsou dostupné předem připravené rozšiřující moduly, či s jakýmikoliv jinými servery s podporou FastCGI.

Systém používá reverzní proxy nginx, které podporou protokolu FastCGI disponuje. Proto je použit volně dostupný modul pro nginx implementující komunikaci tímto protokolem s nezávisle spuštěným daemonem Shibboleth. Tuto komunikaci zajišťuje poslední aplikace *supervisor*. Výsledné blokové schéma lze vidět na obrázku 2. Celý proces autentizace je následující. Uživatel komunikuje s reverzním proxy nginx. Ten je nakonfigurovaný tak, aby vybraná URL překládal přes protokol FastCGI na Shibboleth SP daemona. Po dokončení úspěšné autentizace je uživatel již pomocí proxy překládán na aplikace systému. Těm získaná data o přihlášeném uživateli poskytuje formou HTTP hlaviček.

Obrázek 2: Blokové schéma integrace Shibboleth SP

Kromě modulu pro komunikaci se Shibboleth daemonem (modul *nginx-http-shibboleth*) je použit ještě modul *headers-more-nginx-module*. Ten je nutný pro práci se specifickými HTTP hlavičkami, pomocí kterých jsou přenášena data autentizovaných uživatelů. Oba tyto moduly jsou dostupné pouze ve formě zdrojových kódů. Proto je nutné je před použitím zkompilovat pro konkrétní verzi nginx. Reverzní proxy je, stejně jako jsou ostatní části systému spouštěn v kontejneru. Z toho důvodu byl obdobně jako pro aplikaci generátoru pdf předem vytvořen Docker obraz osahující všechny části vyžadované pro autentizaci s využitím Shibboleth SSO. Nový obraz vychází z oficiálně dostupného obrazu se serverem nginx, který je pro kompilaci nezbytný. Samotné sestavení obrazu se skládá z instalace balíků nutných pro kompilaci (zejména wget, gcc, make a další), stažení zdrojových kódů obou modulů a jejich následné kompilace. Následně jsou již zkompilované moduly přeneseny do nového, konečného obrazu, který opět vychází z oficiálního obrazu nginx. Zde je již instalován Shibboleth SP daemon a supervisor. Dále je připravena adresářová struktura, doplněna konfigurace aplikace supervisor doplněna obecná část konfigurace nginx (zejm. načtení nových modulů), nastaveno oprávnění souborů a složek a nakopírován spouštěcí skript. Tento skript je dále definován jako vstupní bod kontejneru a implementuje periodickou kontrolu stavu všech tří spuštěných aplikací (supervisor, shibboleth daemon a nginx). Protože selhání jediné z nich by znemožnilo správnou funkci, bude v takovém případě zastaven celý kontejner.

Před spuštěním výsledného kontejneru je nutné doplnit konfiguraci Shibboleth

daemona a konkrétní konfiguraci nginx. Konfigurace daemona se skládá z definice mapování atributů, šablony metadat a samotné konfigurace obsahující zejména informace o poskytovateli identit a cestu k certifikátům, kterými je komunikace s IdP šifrována. Po spuštění kontejneru lze na příslušné URL vygenerovat metadata služby. Po jejich zaregistrování u poskytovatele identit je autentizace plně funkční.

4.6.2 Integrace autentizace v aplikacích

Všechny části systému jsou přístupné pouze autentizovaným uživatelům. Proto je Shibboleth SP daemon nakonfigurován tak, aby vyžadoval přihlášení na všech dostupných URL adresách. Výjimku zde tvoří pouze podpůrná administrátorská aplikace Kibana, která neumožňuje integraci autentizace Shibboleth SSO, proto jako jediná implementuje samostatnou správu a autentizaci uživatelů. Ostatní aplikace již předpokládají úspěšně přihlášené uživatele.

Zbylé uživateli dostupné aplikace používají pro rozpoznání identity přihlášeného uživatele speciálně vytvořené HTTP hlavičky. Ty jsou následující:

- cn
- eppn
- epuid
- role

Používané jsou pouze první tři parametry. Atribut *role* obsahuje všechny uživateli přiřazené role oddělené středníkem. Ovšem jedná se o obecné role, jako vyučující, zaměstnanec apod. Nelze tedy těmto informacím přikládat nutnou důvěryhodnost. Naopak při každém požadavku je používán parametr *epuid*. Ten obsahuje v čase unikátní identifikátor uživatele ve formátu „00000@tul.cz“ a slouží ke spárování uživatele s jeho entitou uloženou v databázi. Parametry *cn* obsahující jméno uživatele a *eppn* obsahující e-mail uživatele jsou použity pouze při prvním požadavku, kdy uživatel není ještě zaveden v systému a tato data slouží právě k vytvoření nové entity, která ho bude identifikovat.

4.7 Implementace služeb na pozadí

Jak bylo již popisováno výše, k práci s úlohami na pozadí je primárně využívána knihovna Hangfire. Pomocí ní jsou implementované služby dvou typů. Prvním jsou periodicky opakované úlohy, které se vyvolávají pravidelně, nezávisle na jakékoliv akci. Druhé jsou vytvořeny jednorázově (uživatelem i systémem) a zpracovávány ve frontách. Pro zajištění všech funkcionalit musí nutně tato knihovna perzistentně ukládat informace o stavu workerů, vložených úlohách do front apod. K tomu je opět použita relační databáze PostgreSQL, kde je pro potřeby této knihovny vytvořeno schéma hangfire, kde je odděleně od zbylých dat systému vytvořena samostatná datová struktura, využívaná pouze k těmto účelům.

4.7.1 Typy zpracovávaných úloh

Zpracovávané úlohy na pozadí lze rozdělit do čtyř základních kategorií:

- Notifikace
- Správa uživatelských úkolů
- Správa indexů hledání
- Správa logů

Notifikace

Systém notifikací využívá zejména funkce front. Do fronty se vždy uloží žádost o odeslání konkrétní notifikace. Vždy je uloženo její tělo, předmět/nadpis, úroveň (zda se jedná o informační zprávu, varovnou apod.) a typ providera neboli způsob doručení uživateli. Provider je implementován dvou typů. První doručuje prostřednictvím e-mailu, druhý formou tzv. push notifikací. Při těch je navázána websocketová komunikace s API a jeho prostřednictvím je zaslána zpráva přímo do uživatelova prohlížeče.

Mimo funkce front je využita i funkce opakujících se úloh, které je využíváno pro pravidelné vyhledávání podnětů, v jejichž důsledku jsou odesílány notifikace. V tomto systému se jedná o administrátorskou notifikaci upozorňující na pokles volného diskového prostoru pod nastavenou hranici. V případě, že je definovaná podmínka naplněna, je opět s využitím front zaregistrována nová notifikace všem klíčovým uživatelům, upozorňující na daný fakt.

Jednorázové notifikace jsou generovány buď administrátorem, nebo systémem. Administrátorské notifikace slouží k hromadnému oslovení všech uživatelů vybraným kanálem (e-mailem, push do UI, či oběma současně). Jejich využití je předpokládané zejména k upozornění na odstávku systému apod. Systémové notifikace jsou generovány opět automatizovaně, a to s cílem upozornit na následující fakta:

- Přiřazení nového úkolu
- Zrušení přiřazeného úkolu
- Potvrzení o dokončení úkolu (cílené autorovi daného úkolu)
- Oznámení o změně autorizace do systému

Ukázku vygenerované e-mailové notifikace lze vidět v příloze C.

Správa uživatelských úkolů

Pro správu uživatelských úkolů se využívá výhradně funkce front. Zpracování úlohy je podníceno vždy konkrétní akcí, jako je přihlášení nového uživatele, zadání úkolu pro vyplnění dat studijního programu apod. Obecně lze říci, že existují dva základní typy úloh, a to úloha na zaregistrování nového úkolu/úkolů a na dokončení již existujícího úkolu.

V případě registrace je předávána informace o typu úkolu a identifikátor vázané entity. Tou je v případě autorizace nového uživatele právě *id* tohoto uživatele. V případě vyplnění dat studijního programu je to *id* daného programu. Dle typu úkolu a konkrétní entity se následně vytvoří nový úkol, případně úkoly, nebo skupina úkolů a s využitím předchozích mechanizmů se zaregistrují notifikace, které na tuto skutečnost dané uživatele upozorní.

Úloha pro dokončení úkolu je využívána v případech, kdy byl vytvořen stejný úkol pro více uživatelů. V tom případě po jeho splnění jediným uživatelem jsou v rámci této úlohy přerušeny úlohy ostatním uživatelům. Dále je využívána v případech, kdy úkol není vytvářen administrátorem, ale konkrétním uživatelem. Po jeho dokončení je v průběhu zpracování této úlohy vytvářena notifikace o vyřešení úkolu zpět jejímu autorovi. Jelikož v případě skupinového úkolu je notifikace autorovi zasílána až po dokončení všech dílčích úkolů všemi uživateli, je nezbytné zpracování ve frontě.

Správa indexů hledání

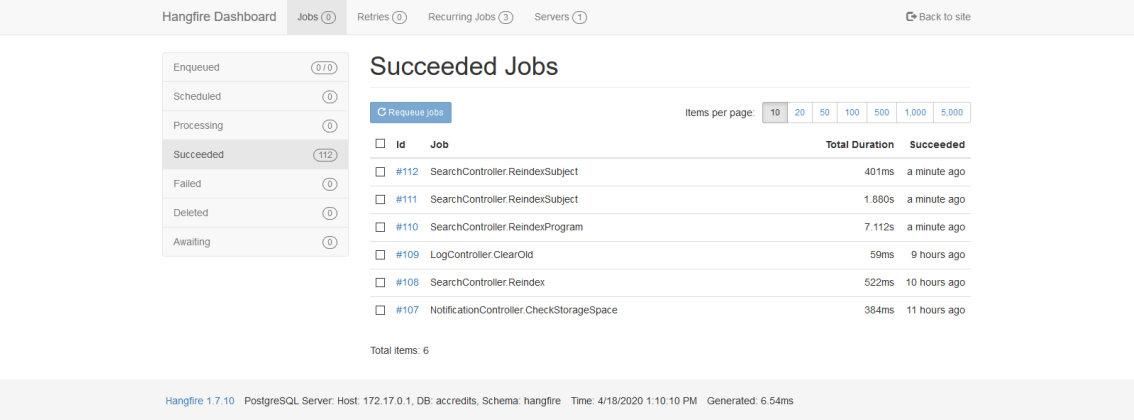
Veškerá indexace dat ve vyhledávací databázi Elasticsearch probíhá výhradně asynchronně na pozadí. Tedy v případě, že uživatel provede změnu indexované entity, se pouze do fronty vloží úloha na její přeindexování a uživatelův požadavek není dále blokován. Přičemž lze přeindexovat předmět, nebo celý studijní program (včetně všech předmětů, které jsou v jeho rámci vyučovány). Pravidelné přeindexování se pak provádí jako opakující se úloha v pevně stanovenou noční hodinu.

Správa logů

Správa logů spočívá v pouhém odstranění starých záznamů z databáze Elasticsearch. Samotné stáří je definováno dle počtu uplynulých dnů od jejich pořízení. Pomocí implementace opakující se úlohy je opět v denních intervalech v pravidelnou noční hodinu sestavován a spouštěn dotaz pro odstranění starých indexů. To je umožněno díky indexování logů do oddělených indexů dle jejich data pořízení. Snadno tak lze smazat všechny log záznamy vybraných dní, tedy i těch pořízených před definovaným datem, které označujeme za „staré“.

4.7.2 Monitoring služeb

Pro monitorování stavu zpracovávaných úloh je spuštěna samostatná webová aplikace využívající připraveného uživatelského rozhraní dostupného také v knihovně Hangfire. Ukázku lze vidět na obrázku 3. Konkrétně lze vidět jednoduché statistiky o proběhlých úlohách, zobrazit chybová hlášení těch, co selhali či zkontrolovat opakující se úlohy, nebo i samotné workery. Dále je možné danou úlohu odstranit z fronty nebo spustit (i opakovaně). Tyto úkony lze provádět i s periodicky spouštěnými úlohami. Aplikace implementuje autentizaci pomocí Shibboleth SSO standardním způsobem. Jelikož se jedná o administrátorský nástroj, je služba dostupná pouze uživatelům s touto rolí přiřazenou v systému.



Obrázek 3: Ukázka webového rozhraní pro monitoring služeb na pozadí

4.8 Implementace logování

Aplikační logy jsou často jediným způsobem, jak odhalit a diagnostikovat chyby v systému či například pouze sledovat chování uživatelů. Z toho důvodu budou i aplikace tohoto systému zaznamenávat prostřednictvím logů dění v systému.

4.8.1 Zaznamenávaná data

Obsah, a tedy i formát logované zprávy, se liší dle typu zdrojové aplikace. První typ logů vytváří reverzní proxy nginx, přes kterou prochází všechny požadavky uživatelů. Ta vytváří pro každou aplikaci dva log soubory. První je typu *access* a obsahuje informace o všech požadavcích na danou aplikaci. Druhý je typu *error*, který obsahuje hlášení o chybách, jakými může být nedostupná cílová aplikace apod.

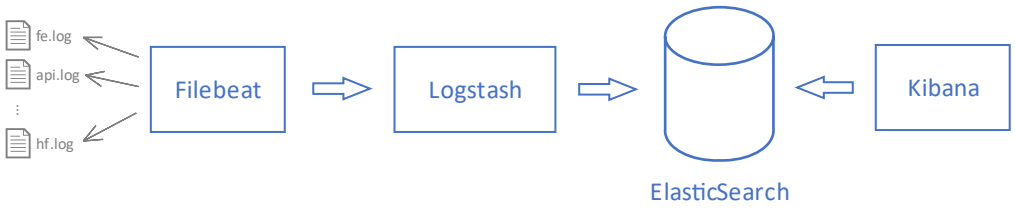
Všechny aplikace vyvíjené na platformě .NET Core implementují jednotný formát logů. Ty mají nadefinované dva typy výstupů. Prvním je terminál aplikace. Druhým je textový soubor, který se automaticky dělí dle kalendářních dnů. Do souborů jsou ukládány všechny zprávy všech úrovní ze všech zdrojů. Na terminál se vypisují pouze „ručně“ vytvářená hlášení a informační zprávy o životním cyklu aplikace (informace o úspěšném spuštění apod.).

Ostatní aplikace vyvíjené v JavaScriptu, kterými jsou Frontend a PdfCreator již log soubory negenerují. Důvody jsou následující. Aplikace Frontend je SPA webová aplikace spouštěná v prohlížeči uživatele, která sama žádné soubory generovat nemůže. Řešením by mohlo být, zachycená hlášení o chybách odesílat na server, kde by byla archivována. Schopnost danou chybu odeslat ale není vždy zajištěna, a to například z důvodu výpadku sítě apod. V případě aplikace PdfCreatoru jsou vypisována pouze pomocná hlášení do terminálu, která byla nápomocna při ladění aplikace. Jelikož ale veškerá komunikace s touto aplikací je iniciována z API a hlášení o chybě vzniklé při generování pdf je vráceno v těle odpovědi, jsou informace o případných chybách logovány přímo zde.

4.8.2 Centralizovaný sběr logů

Protože pro poskytnutí všech funkcionalit systému je nutný současný provoz osmi různých kontejnerů s různými aplikacemi, potenciální zkoumání logů jednotlivých souborů by mohlo být v některých případech obtížné. Zejména v případech, kdy je sledovaná akce klienta zpracovávána různými aplikace a je nutné tento tok sledovat. Proto je implementováno centralizované logování, které umožní pohodlné prohledávání log záznamů všech aplikací na jediném místě.

K tomu je použit již zmiňovaný ELK stack, kde Elasticsearch slouží jako úložiště pro všechny záznamy, Logstash zpracovává vstupní log záznamy a Kibana slouží opět pouze k vizualizaci uložených dat. V případě našeho systému je tato trojice ještě rozšířena o aplikaci Filebeat, která načítá konkrétní log soubory a nové záznamy odesílá ke zpracování do Logstash. Celé blokové schéma lze vidět na obrázku 4.



Obrázek 4: Blokové schéma centralizovaného sběru aplikačních logů

Celý proces sběru logů je následující. Jednotlivé aplikace vytvářejí své log soubory. Ty jsou perzistovány na disk serveru tak, aby při restartu kontejneru nedošlo k jejich ztrátě. Tyto soubory jsou však archivovány pouze po dobu dvou dní. To je pouze minimální doba, která slouží k pokrytí případné krátkodobé nedostupnosti centrálního úložiště. Filebeat spuštěný v samostatném kontejneru je nastaven tak, aby sledoval změny v jednotlivých souborech a data odesílal do aplikace Logstash, běžící rovněž v samostatném kontejneru. Přičemž jednotlivým záznamům přiděluje příznaky značící o jaký soubor se jedná, jméno zdrojové aplikace a jméno běhového prostředí. Současně spolu tyto dvě aplikace vytvářejí frontu, která je schopna pokrýt větší zátěž systému bez ztráty záznamů.

Aplikace Logstash je nakonfigurována pro zpracování všech typů vstupních logů, které rozpoznává pravě dle přidělených příznaků. Tzn. zejména naparsování vstupního řetězce do dílčích vlastností, dle kterých bude možno dále filtrovat (typicky úroveň log zprávy, čas vzniku atp.) a odstranění původního (plného) znění zprávy. Na logy o přístupech uživatelů vytvářené reverzním proxy nginx jsou aplikovány další moduly *useragent* a *geoip*. Ty získaná data transformují do nových informací, díky kterým lze provádět později další analýzy o uživatelích a jejich chování v systému. Primárně se jedná o informace o webovém prohlížeči a poloze uživatele. Tato výstupní data jsou již odesílána do databáze Elasticsearch, kde jsou zaindexována pro potřeby dalšího hledání.

Pro zamezení plnění disku starými záznamy je nutné periodicky odmazávat staré

neaktuální záznamy. Touto funkcionalitou však samotný Elasticsearch nedisponuje. Proto je na BE implementována úloha opakující se s periodou jednoho dne, která pomocí konektoru NEST maže z databáze logy starší šedesáti dnů.

4.9 Nasazení systému do produkčního prostředí

Po implementaci celého systému je nutné provést jeho nasazení do produkčního prostředí, kde bude dostupný všem cílovým uživatelům. Pro zjednodušení administrátorských úkonů, při následných aktualizacích, je část tohoto procesu automatizována. Jedná se primárně o nasazení jednotlivých aplikací, u kterých na rozdíl od databázového serveru lze předpokládat nutnost aktualizací vyžadující jejich přenasazení.

4.9.1 Příprava prostředí

Před zahájením automatizovaných procesů je nutné připravit běhové prostředí. To se skládá z konfigurace firewallových prostupů serveru, vytvoření požadované adresářové struktury (pro perzistenci systémových dat), nahrání certifikátů a instalace potřebných balíků. Jedná se zejména o instalaci kontejnerizačního nástroje Docker a databázového serveru PostgreSQL, které jsou pro běh systému nezbytné.

4.9.2 Funkce CI/CD

Všechny zdrojové kódy jsou verzovány s pomocí nástroje Git v systému GitLab, který je provozován interně v síti TUL. Systém GitLab nabízí funkce Continuous integration a Continuous delivery, tedy funkce tzv. Průběžné integrace a Průběžného dodání, pomocí kterých lze automatizovat procesy. S využitím těchto funkcí je možné definovat (pomocí skriptu) tzv. pipeline, ve které se definují jednotlivé kroky, které se vykonávají zcela automatizovaně po zachycení vybrané události. Tou bývá zpravidla změna ve zdrojovém kódu.

Aby však bylo možné tyto funkce využít, je nezbytný tzv. runner. Tj. služba běžící na serveru, která vykonává definované skripty v rámci pipeline. Verzovací systém tedy pouze řídí spouštění těchto úloh, ale vykonávány jsou v prostředí konkrétního runneru. Těch může být obecně více a mohou být i sdíleny napříč různými projekty. Výběr konkrétního runneru pak bývá zcela náhodný (dle aktuální dostupnosti apod.). Je ale i možné specifikovat vykonávání konkrétních úloh pouze vybranými runnery. Snadno tak lze například oddělit odkud probíhá nasazení aplikace dle cílového běhového prostředí (testovací, produkční atd.).

Automatizační pipeline implementovaného systému je spouštěna automaticky při každé změně jakéhokoliv zdrojového souboru v jakékoliv vývojové větvi a skládá se celkem z následujících čtyř kroků:

- build_test
- cleanup_build_test

- deploy_prod
- cleanup

Pro kroky build_test a cleanup platí, že jsou spouštěny vždy. Spuštění zbylých dvou (tedy cleanup_build_test a deploy_prod) je dále podmíněno. Pro všechny ale platí, že provádí operace nad Docker kontejnery.

build_test

Prvním krokem je build_test. Ten představuje základní prvek CI. Jeho cílem je otestovat validitu kódu. Toho je docíleno přeložením všech aplikací systému a případného spuštění automatizovaných testů, kterými lze i ověřit kromě syntaktické správnosti i tu sémantickou.

cleanup_build_test

Tento krok je spouštěn pouze v případě, že předchozí selže. Jeho jediným cílem je provést odstranění kontejneru, ve kterém probíhala validace (viz. předchozí krok), a který byl „násilně“ ukončen z důvodu nalezené chyby v programu.

deploy_prod

Krok deploy_prod slouží k plně automatizovanému nasazení všech aplikací (resp. kontejnerů) do produkčního prostředí. Proto je aktivován pouze v případě, že akci vyvolala změna ve větvi *master* (tj. jediná větev, ze které je dovoleno nasazení aplikace na produkční server). Součástí tohoto kroku je zastavení všech produkčních kontejnerů, sestavení nových (z nových zdrojových kódů) a jejich spuštění.

Kromě kontejnerů s trvale spuštěnými aplikacemi je v tomto kroku jednorázově spouštěna i aplikace (v kontejneru) pro přípravu prostředí. Tj. speciálně vyvinutá aplikace, která slouží pro zajištění základních předpokladů pro běh systému. Konkrétně se jedná o provedení vygenerovaných migrací databáze PostgreSQL (včetně přípravy struktury využívanou službou Hangfire) a registraci vytvořených periodických úloh zpracovávaných na pozadí.

cleanup

Poslední krok je spouštěn opět vždy a slouží k vyčištění všech již nepotřebných obrazů kontejnerů, které byly vytvořeny v předchozích krocích.

4.9.3 Konfigurace verzovacího systému

Jelikož systém GitLab provozovaný na TUL nedisponuje žádným sdíleným runnerem, který by bylo možné použít, bylo nutné připravit vlastní. Proto byla na produkčním serveru tato služba nainstalována a tzv. registrována ke konkrétnímu projektu vedenému v systému GitLab pomocí registračního tokenu. Při konfiguraci bylo nutné nastavit exekutora neboli vykonavatele. V podstatě se jedná o nastavení

cílového prostředí, kde jsou spouštěny skripty definované v pipeline. K dispozici je například vzdálený server připojený přes protokol SSH, virtualizovaný systém v nástroji VirtualBox, či Docker. V případě tohoto systému je ale použita nejpřímější volba pro spouštění skriptů přímo v prostředí serveru, kde je runner registrován, tedy na běhovém produkčním prostředí. Díky tomu je umožněno snadno, na jediném serveru, spouštět úlohy průběžné integrace, které přímo neovlivňují publikované aplikace, tak i snadno nové verze publikovat.

5 Testování

Testování bývá důležitou částí vývoje informačního systému. Obecně testy slouží k ověření správné funkčnosti systému a existuje jich několik typů.

5.1 Jednotkové testy

Jednotkové testy umožňují kontrolovat správnost algoritmů na nejnižší úrovni, tzv. jednotek. To představuje otestování vybraných kusů kódu, jako jsou jednotlivé metody implementovaných služeb apod. Implementace těchto testů spočívá v pouhém „zavolání“ implementované metody a následné kontrole výstupních dat. Ta zpravidla spočívá v porovnání návratové hodnoty s očekávanou (dle zadaného vstupu), či ověření, že v průběhu zpracování nebyla vyvolána výjimka. Díky jednoduchosti těchto testů je lze snadno spouštět zcela automatizovaně, například v rámci rutin průběžné integrace.

V našem systému jsou implementovány jednotkové testy s použitím NUnit. To je jeden z testovací frameworků pro jazyk C#. Existují i další, jako je novější XUnit. Jejich základní funkce jsou ale velmi podobné a pro potřeby testování implementovaného systému zcela marginální. V našem případě slouží k otestování služeb na backendu, které by se jinak obtížně ladily. Protože ale služby na BE často využívají front a data přenášejí přes databázi, je nutné zavést postupy, které toto umožní. Pro potřeby těchto testů nebude využívána standardní (perzistentní) databáze, ale pouze její „InMemory“ obdoba. Pro každý test bude automaticky vytvořena prázdná databáze v paměti (dle definovaného schématu). Do ní budou uložena potřebná data pro otestování konkrétní funkcionality a následně bude celá databáze opět smazána. Tyto databáze mají i svá omezení, jako je chybějící podpora cizích klíčů a dalších specifik, kvůli kterým není vhodná pro otestování databázových operací, ale pro tyto potřeby je dostačující.

5.2 Integrační testy

Integrační testy slouží pro otestování komunikace mezi více službami, tedy jejich tzv. integraci. Oproti jednotkovým testům bývá tento typ obtížněji automatizován, protože je třeba spustit tyto aplikace v „testovacím“ režimu a zajistit prostupnost požadovaného komunikačního kanálu.

Zejména z těchto důvodů byla integrace jednotlivých aplikací systému testována v průběhu vývoje bez použití automatizačních technik. K tomu byl využit opět kontejnerizační nástroj Docker, s jehož pomocí bylo možné spustit a testovat jednotlivé aplikace v prostředí velmi podobném tomu produkčnímu. Integraci s aplikacemi třetích stran lze však v některých případech testovat i v rámci jednotkových testů. Příkladem je vytvořený jednotkový test pro odeslání notifikace, který krom správné funkce služby testuje i integraci se SMTP serverem, který zprostředkovává odeslání této zprávy.

5.3 Testy uživatelského rozhraní

I pro testy uživatelského rozhraní existují nástroje, kterými lze tyto činnosti automatizovat (bližší srovnání deseti nejpoužívanějších lze nalézt v [9]). Ty se i s použitím technik strojového učení snaží o automatizovanou tvorbu testovacích scénářů a výstupních reportů.

Testování uživatelského rozhraní implementovaného systému proběhlo výhradně ručně, a to ve dvou fázích. První fáze představuje kontinuální testování v průběhu vývoje aplikace. To představovalo především kontrolu správné funkce transpilovaného kódu napříč různými webovými prohlížeči a případné optimalizace zdrojového kódu, či konfigurace zmiňované transpilace (např. integrací již zmiňovaných polyfills). Ve druhé fázi pak probíhalo finální testování již vytvořeného systému. Jeho cílem bylo otestovat několika uživateli s různými uživatelskými rolemi předem definované postupy. Ty byly vytvořeny tak, aby obsáhly všechny požadované funkcionality systému. Výsledky této fáze představovaly náměty na úpravy uživatelského rozhraní, které by zlepšily, nebo jinak usnadnili práci koncovým uživatelům, či ukázaly další chyby (týkající se zejm. řízení oprávnění), které bylo nutné bezpodmínečně opravit.

6 Závěr

Práce byla primárně zaměřena na návrh a následnou implementaci informačního systému pro správu akreditačních spisů, přičemž požadované funkce tohoto systému vycházejí z provedené analýzy. Předně se jedná o funkce umožňující definici nového studijního programu a reakreditaci stávajícího, a to s možností „zkopírování“ definic vyučovaných předmětů z jiných spisů. Dále byly kladeny požadavky na trvalé uložení stavu akreditace, fulltextové vyhledávání v datech a podporu uživatelských úkolů, které by společně s automatickými notifikacemi měly zefektivnit komunikaci mezi spolupracujícími akademickými pracovníky. Samozřejmostí bylo vytvoření šablon pro export výstupů do souborů typu pdf.

Výsledkem práce je realizovaný informační systém, který splňuje všechny body zadání. Jeho jednotlivé aplikace jsou s využitím kontejnerizačního nástroje Docker nasazeny do produkčního prostředí na Technické univerzitě v Liberci. Toto nasazení, je včetně případné migrace (úpravy) databázové struktury a spouštění jednotkových testů plně automatizováno s využitím funkcí verzovacího systému GitLab, ve kterém jsou ukládány všechny zdrojové kódy. Vyjma těchto předpokládaných funkcionalit jsou implementovány další nástroje určené pro administrátory systému umožňující jednodušší správu. Jedná se především o centralizované logování a jednoduchou webovou aplikaci pro monitoring a základní ovládání úloh zpracovávaných na pozadí.

Systém nabízí i další možná rozšíření do budoucna. Jedním z nejzásadnějších by mohlo být rozšíření systému o podporu zbylých listů akreditačního spisu tak, aby bylo možné sestavit a následně vygenerovat kompletní spis k akreditaci daného studijního programu. Dále by to mohlo být například rozšíření fulltextového vyhledávání pro zajištění vyhledávání dle dalších datových polí či začít indexovat obsahy archivovaných studijních programů. Také by bylo možné rozšířit systém úkolů o další typy notifikací, které by umožňovaly ručně definovat příjemce a více personalizovat odesílané zprávy.

Použitá literatura

[1] *Metodické materiály. NAÚ* [online]. Praha: Národní akreditační úřad pro vysoké školství, c2020 [cit. 2020-02-22]. Dostupné z: https://www.nauvs.cz/index.php/cs/metodiky

[2] NOLL, Jan. *Systém pro správu akreditačních spisů*. Liberec, 2019. Magisterský projekt. Technická univerzita v Liberci, Fakulta mechatroniky, informatiky a mezioborových studií.

[3] *The State of the Octoverse: Top languages* [online]. GitHub, c2019 [cit. 2020-05-08]. Dostupné z: https://octoverse.github.com

[4] *MySQL vs PostgreSQL... Okta* [online]. Praha: Krasimir Hristozov, 2019 [cit. 2020-02-22]. Dostupné z: https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres

[5] *Foreword: What does "micro" mean?* [online]. Pallets, c2010 [cit. 2020-05-18]. Dostupné z: https://flask.palletsprojects.com/en/1.1.x/foreword/#what-does-micro-mean

[6] *Explore - Docker Hub* [online]. San Francisco: Docker, c2020 [cit. 2020-05-09]. Dostupné z: https://hub.docker.com/search?q=&type=image

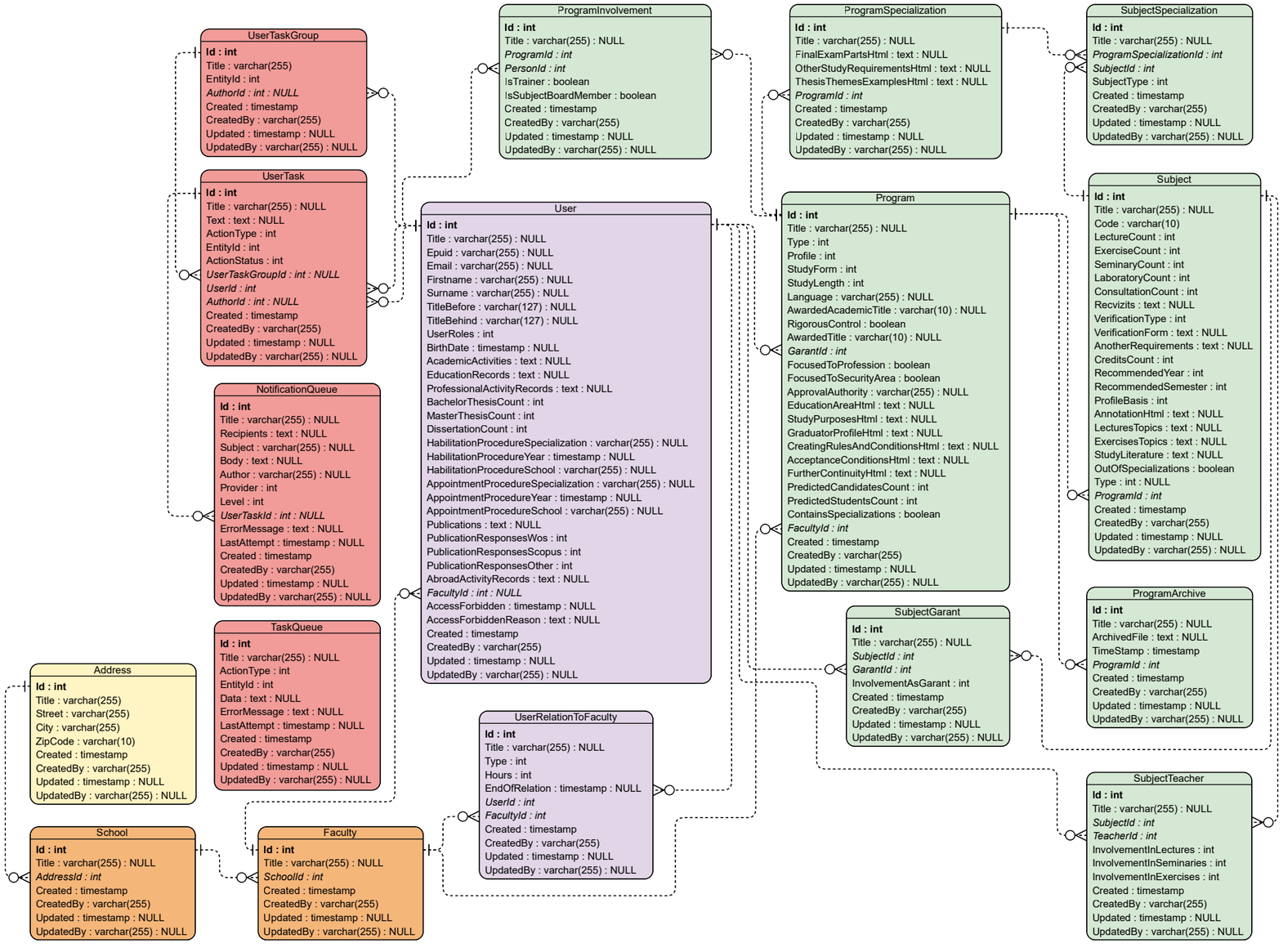
[7] *NuGet Gallery | Home* [online]. Redmond: Microsoft, c2020 [cit. 2020-05-10]. Dostupné z: https://www.nuget.org

[8] *Service Provider (SP). Czech academic identity federation eduID.cz* [online]. Praha: CESNET, 2020 [cit. 2020-04-14]. Dostupné z: https://www.eduid.cz/cs/tech/sp

[9] *Best Automation Testing Tools for 2020 (Top 10 reviews)* [online]. Brian, 2017 [cit. 2020-05-14]. Dostupné z: https://medium.com/@briananderson2209/best-automation-testing-tools-for-2018-top-10-reviews-8a4a19f664d2

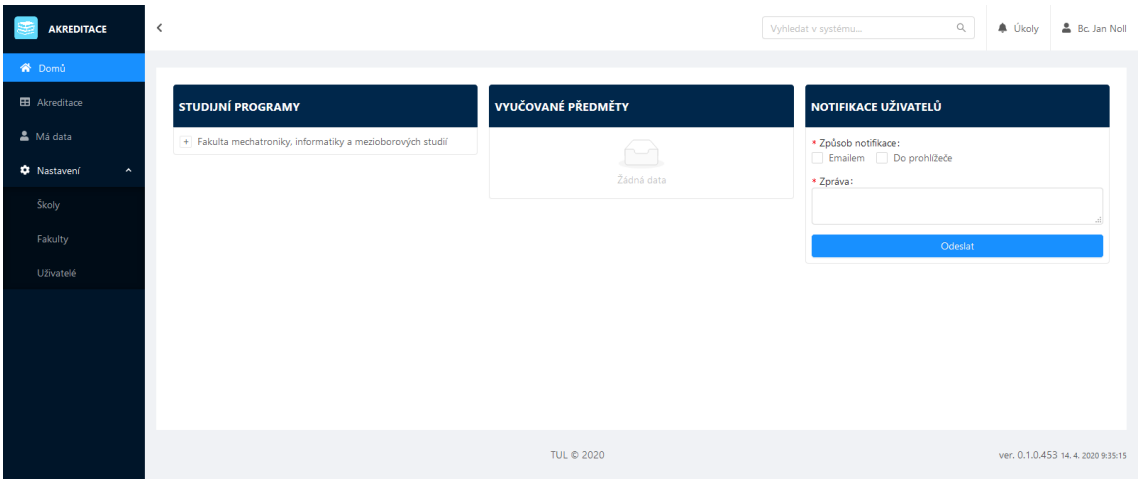
Přílohy

A Struktura databáze

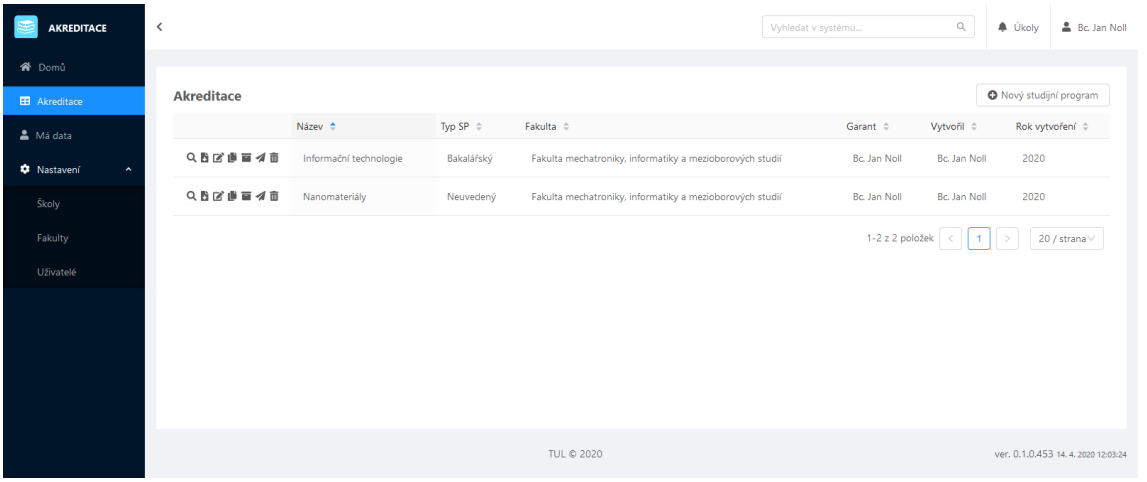


Obrázek 5: Struktura navržené databáze

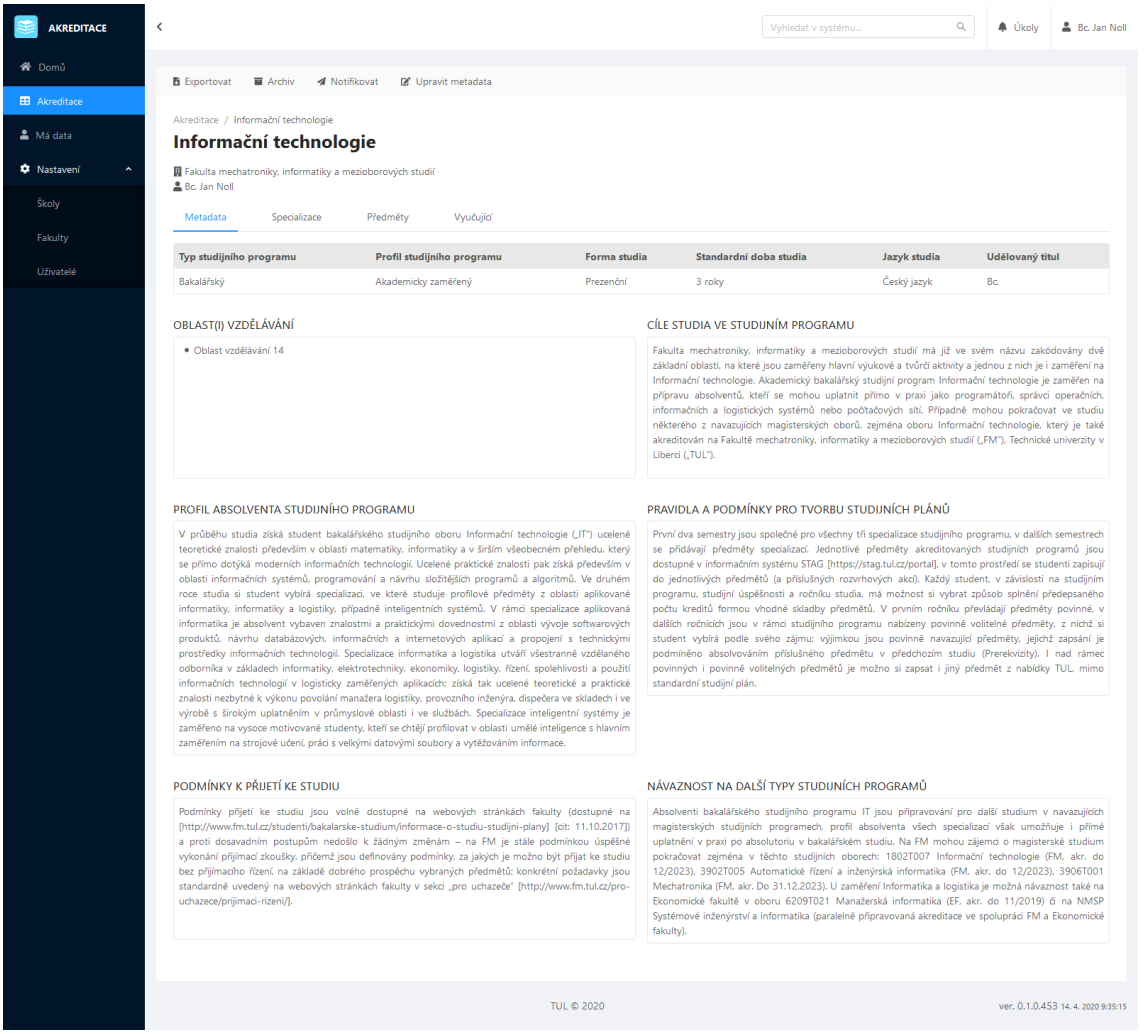
B Výsledná podoba uživatelského rozhraní



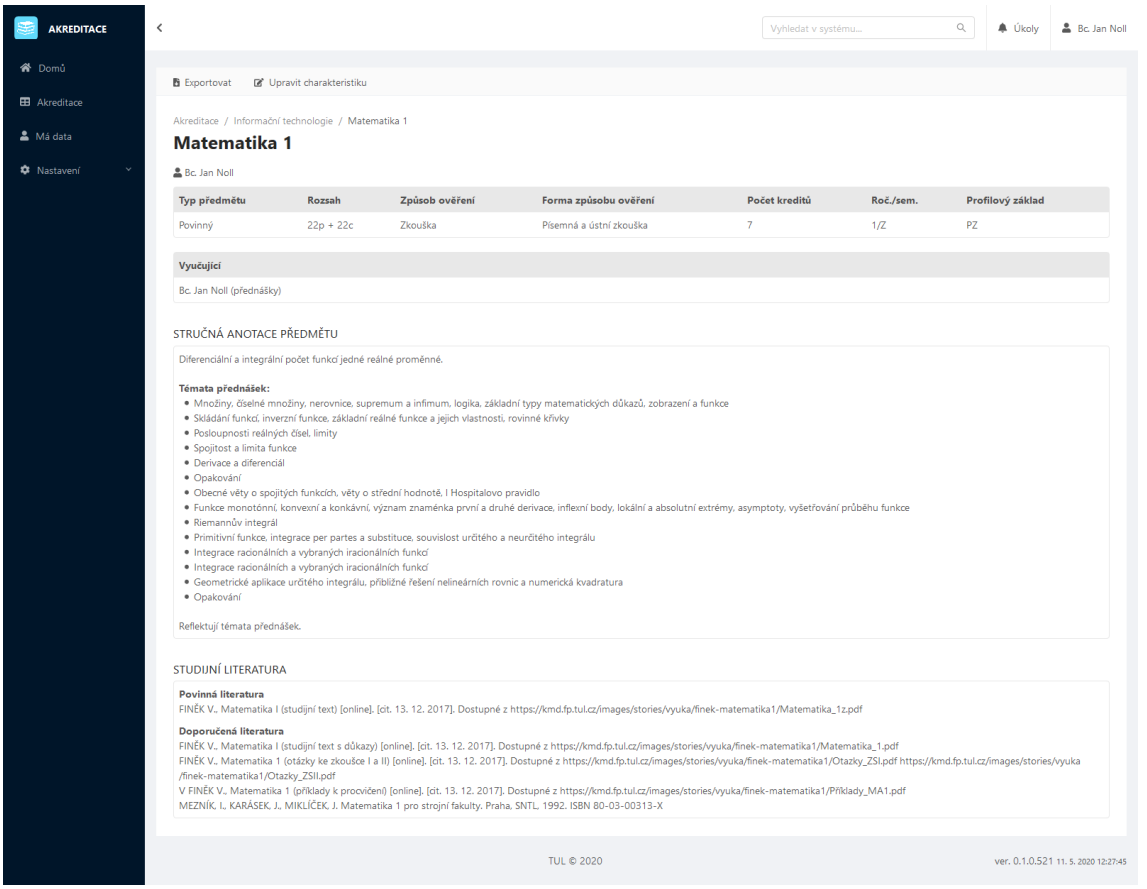
Obrázek 6: Výsledná domovská stránka



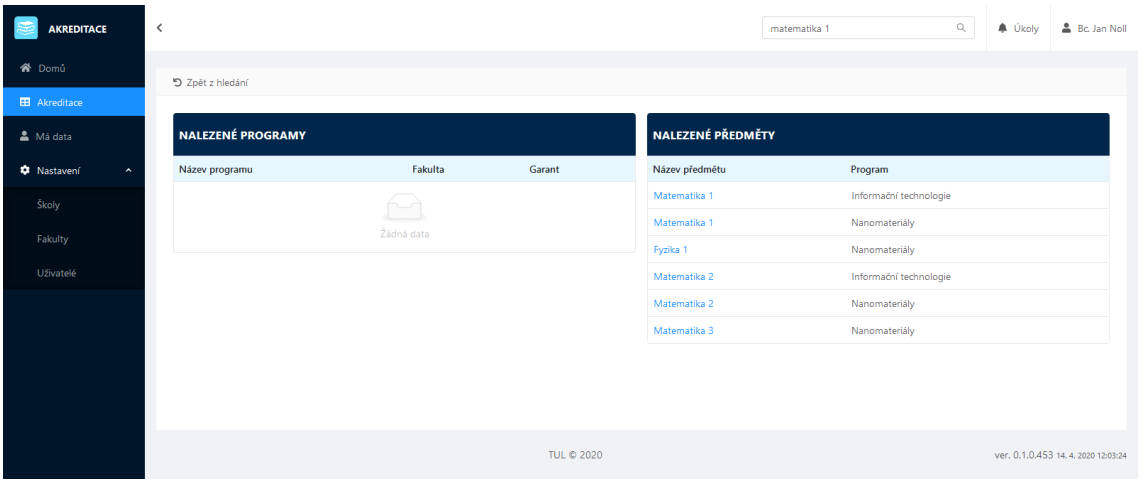
Obrázek 7: Výsledná stránka seznamu programů



Obrázek 8: Výsledná stránka detailu programu

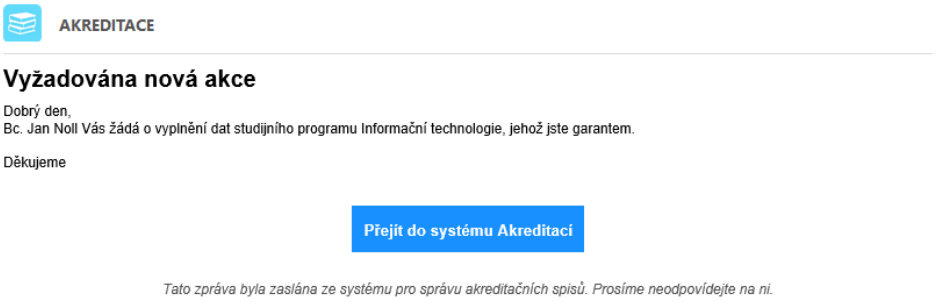


Obrázek 9: Výsledná stránka detailu předmětu



Obrázek 10: Výsledná stránka s výsledky hledání

C Výsledná podoba generovaných e-mailů



Obrázek 11: Ukázka e-mailu o vytvořeném úkolu