Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Systém pre dlhodobú správu zadaní úloh

Bakalárska práca

2023

Filip Piták

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Systém pre dlhodobú správu zadaní úloh

Bakalárska práca

Študijný program:	Aplikovaná Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	Mgr. Pavel Petrovič, PhD.

Bratislava, 2023
Filip Piták

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Filip Piták
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Systém pre dlhodobú správu zadaní úloh
System for Long-Term Management of Tasks

Anotácia: Viac ako desať rôznych predmetov v študijnom programe Aplikovaná informatika využíva systém LIST, v ktorom sú uložené zadania úloh z viac ako desiatich rokov, zostavy úloh, automatické testy, systém umožňuje rozdelenie študentov do skupín zverejňovanie materiálov, hodnotenie, automaticky vytvára tabuľky hodnotení, poskytuje automatickú a manuálnu spätnú väzbu študentom a prispieva tak k zlepšovaniu kvality výučby. Systém je vyvinutý vo webovom frameworku, ktorý za dobu jeho prevádzky morálne zastaral a skôr alebo neskôr bude portebné vytvoriť celkom nový systém.

Cieľ: Cieľom tejto práce je navrhnúť a položiť dobré základy nového systému. Vzhľadom na rozsah systému LIST je ťažké predpokladať, že nová verzia vznikne v rozsahu jednej bakalárskej práce, ale mal by vzniknúť funkčný a používateľný základ, ktorého kód bude ľahko udržiavateľný, modulárny, dobre dokumenovaný a prehľadný tak, aby sme postupným rozširovaním systému mohli získať plnohodnotnú náhradu systému LIST.

Literatúra: Andrej Jursa: Nový dlhodobý viacúčelový sklad zadaní, bakalárska práca, FMFI UK Bratislava, 2013.
Peter Jurčo: Dlhodobý viacúčelový sklad úloh na cvičenia, bakalárska práca, FMFI UK Bratislava, 2009.
Duldulao D.B, Villafranca S.: Full-Stack Web Development with Spring Boot and Angular: A practical guide to building your full-stack web application, Packt Publishing, 2022.

Kľúčové slová: webová aplikácia, angular, spring, lms

Vedúci: Mgr. Pavel Petrovič, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 05.09.2022

Dátum schválenia: 17.10.2022

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

Čestné prehlásenie: Čestne prehlasujem, že som túto prácu vypracoval samostatne iba s použitím uvedených zdrojov a na základe konzultácií so školiteľkou.

. . . .

Poďakovanie: Poďakovanie patrí školiteľovi Mgr. Pavelovi Petrovičovi, PhD. za jeho ochotný prístup, rady, usmernenia a čas, ktorý mi venoval počas tvorby bakalárskej práce.

Chcel by som poďakovať aj spoločnosti PosAm, s.r.o. za aktívne sprostredkovanie nových znalosti a zručností, ktoré sa využili v bakalárskej práci. Tiež by som chcel poďakovať kolegyni Mgr. Maríne Madovej zo spoločnosti PosAm, s.r.o. za cenné rady, ochotnú pomoc pri návrhu architektúry a implementácii projektu.

Abstrakt

Štúdium informatiky sa stáva obľúbeným odborom a pre veľké množstvo študentov znamená efektívny a nápomocný systém pre podporu vyučovania ako nevyhnutný. Na fakulte Matematiky, Fyziky a Informatiky v Bratislave preto v roku 2013 vznikol systém LIST (Long-term Internet Storage of Tasks) využívaný primárne ako webový portál na zverejnenie zadaní kurzov, odovzdávanie a automatické ohodnotenie riešení pre programátorske úlohy. LIST spĺňa potreby pre priebeh vyučby ale obsahuje nedostatky z hľadiska návrhu, UI/UX a možnosť rozšírenia.

Vytvorili sme podrobnú analýzu a návrh pre nový nahrádzajúci systém LIST -NG (LIST– New Generation) s použitím moderných architektonických prístupov a technológií. Architektúru systému tvorí 12 škálovateľných modulov implementované pomocou framework-u Spring s používateľským rozhraním postavená ako Angular webová aplikácia. V návrhovej časti práce sme podrobne vypracovali popis architektúry, modelu systému, vzhľad používateľského rozhrania a infraštruktúry prostredia pre nasadenie LIST-NG. Realizovali sme implementáciu potrebných časti pre zabezpečenie komunikácie medzi modulmi, spracovanie autentifikácie používateľa s prístupmi, definíciu štruktúry jednotlivých projektov s ohľadom na modularitu a funkcionalitu, zdieľané funkcionality a automatizácie pre všetky moduly. Úspešne sme pokryli funkcionalitu v 6-ich moduloch, ktoré slúžia aj ako dokumentácia a návod zapojenia mnoho technológií do jedného celku. Taktiež sme pre používateľské rozhranie vytvorili definíciu efektívnej a prehľadnej štruktúry, množstvo konfigurovateľných komponentov a implementácie tvoriace jadro logiky a mechanizmu aplikácie. Táto práca bude aj slúžiť ako smernica pre ďalšie etapy vývoja inými študentmi.

Kľúčové slová: learning management system, webový portál, LIST-NG, modularita, moderné technológie.

Abstract

Computer science is becoming a popular field of study and a vast amount of students implies that an effective and helpful tool supporting the needs of teaching is essential. Therefore, at the Faculty of Mathematics, Physics and Informatics in Bratislava in 2013 a new system LIST (Long-term Internet Storage of Tasks) has been created, which is used primarily as a web portal for publishing assignments, submitting solutions and automatically evaluating solutions of programming tasks. LIST meets the needs for managing the course of schooling, but yet contains a few flaws in terms of design, UI/UX and the possibility of further development.

We have created a detailed analysis and design for the new system LIST-NG (LIST-New Generation) with the use of modern architectural approaches and technologies. The system architecture consists of twelve scalable modules implemented with the Spring framework and a user interface built as an Angular web application. In the design part of the document we developed a description of the architecture, system model, user interface design and the infrastructure of the target environment for deploying LIST-NG. We implemented the necessary parts to secure communication between modules, process user authentication with his access, define the general structure of modules while taking into account the systems, functionality, shared functionalities and automation's for all modules. We successfully developed some functionality in 6 different modules, which may be used as a documentation and guideline for using many technologies in a single whole. As well for the user interface, we created an effective and clean definition of the project structure, many reusable and configurable components and implementations of the domains core logic and mechanisms. This document shall also be used as a guideline for future development by other students.

Keywords: learning management system, web portal, LIST-NG, modular, modern technologies.

Obsah

Úvod 1

1 Východiská práce 2

1.1 Teoretické východiská 2

1.1.1 Learning management system 2

1.1.2 Domain-Driven Design 3

1.1.3 Modul a modularita 3

1.1.4 Hexagonálna architektúra 4

1.1.5 Mikroservisná architektúra 4

1.1.6 Unit testing 5

1.2 Existujúce systémy 6

1.2.1 Moodle 6

1.2.2 Long-term Internet Storage of Tasks 7

1.3 Technológie 12

1.3.1 Object-Relational Mapping 12

1.3.2 Hibernate ORM 12

1.3.3 REpresentational State Transfer 13

1.3.4 Spring 14

1.3.5 Consul 14

1.3.6 Azure Kubernetes 15

1.3.7 Feign 16

1.3.8 Angular 16

1.3.9 Bootstrap 17

1.3.10 Angular Material 17

1.3.11 Thymeleaf 18

1.3.12 Nginx 18

2 Návrh systému 19

2.1 Požiadavky a očakávania 19

2.1.1 Popis systému 19

2.1.2 Požiadavky 20

2.2 Analýza 22
2.2.1 Technológie 22
2.2.2 Modularita 23
2.2.3 Doménový model 27
2.3 Finálny návrh systému 30
2.3.1 Architektúra 30
2.3.2 Model systému 32
2.3.3 Používateľské rozhranie 36
2.3.4 Infraštruktúra 45

3 Realizácia a implementácia **47**
3.1 Doména a služby 47
3.1.1 Štruktúra projektu služby 47
3.1.2 Implementácia zdieľanej knižnice 49
3.1.3 Consul 50
3.1.4 Controller a mapovania 51
3.1.5 Komunikácia medzi službami 52
3.1.6 Vytvorenie a odosielanie emailov 53
3.1.7 Modularita v REST API Gateway 55
3.1.8 Autentifikácia 55
3.1.9 Základ controller-ov v gateway 60
3.1.10 Oprávnenia v kurzoch 61
3.2 Používateľské rozhranie 62
3.2.1 Štruktúra projektu 62
3.2.2 Bezpečnosť a autorizácia prístupu 63
3.2.3 Moduly a routing 64
3.2.4 Lokalizácia textov 65
3.2.5 Identita stránky 66
3.2.6 Filtrovanie 67
3.2.7 Všeobecná tabuľka pre správu 69
3.2.8 Ošetrovanie chýb 69
3.3 Infraštruktúra prostredia 71
3.3.1 Príprava a inštalácia 71

Záver **73**

Príloha A **78**

Zoznam obrázkov

1.1 Skladba mikroservisnej architektúry . . . 5
1.2 Konfiguračný formulár v LIST-e . . . 9
1.3 Navigácia v učiteľskom rozhraní LIST-u . . . 10
1.4 Architektúra Kubernetes cluster . . . 15

2.1 Prvá iterácia návrhu architektúry . . . 24
2.2 Druhá iterácia návrhu architektúry . . . 25
2.3 Tretia iterácia návrhu architektúry . . . 26
2.4 Prvá iterácia návrhu domény . . . 27
2.5 Druhá iterácia návrhu domény . . . 29
2.6 Finálny návrh domény . . . 35
2.7 Všeobecné rozloženie obsahu aplikácie . . . 37
2.8 Okno pre prihlásenie . . . 37
2.9 Okno pre žiadosť obnovy hesla . . . 38
2.10 Okno pre obnovu hesla . . . 39
2.11 Domovské okno aplikácie . . . 40
2.12 Všeobecné rozloženie okna so správou . . . 41
2.13 Vytvorenie zostavy úloh . . . 42
2.14 Prehľad kurzu - Úlohy . . . 43
2.15 Prehľad kurzu - Skupiny . . . 44
2.16 Prehľad kurzu - Projekty . . . 45

3.1 Štruktúra projektu služby . . . 48
3.2 Vzhľad formátovaného emailu . . . 54
3.3 Domovská stránka LIST-NG . . . 66
3.4 Konfigurované modálne okno filtrovania . . . 68
3.5 Okno pre správu používateľov . . . 69
3.6 Chybové okno 404 . . . 70
3.7 Modálne okno pre zobrazenie chýb . . . 71

Zoznam implementačných ukážok

1.1 Ukážka využitia ORM mapovania v LIST. 11
1.2 Ukážka Hibernate anotácií. 13
3.1 Vzor repository rozhrania pre vykonanie CRUD operácii nad databázou. 49
3.2 Spustiteľná trieda služby. 49
3.3 Vzor API rozhrania s definíciou mapovaní a ich implementácii. 52
3.4 Implementácia proxy pomocou Feign klienta pre komunikáciu so službou. 53
3.5 Generovanie HTML template-ov a ich odoslanie emailom. 54
3.6 Nastavenie konfigurácie Spring Security autentifikácie. 57
3.7 Servisná trieda pre authentifikáciu používateľa. 58
3.8 Úkažka servisnej triedy pre operácie s JWT tokenom. 59
3.9 Načítanie používateľa z kontextu dopytu. 61
3.10 Konfigurácia routing ciest. 64
3.11 Asynchrónna lokalizácia textov s príkladom využitia v HTML template. 65
3.12 Ukážka prepísania štýlov komponenty z Angular Material. 67
3.13 Ukážka konfigurácie filtrov. 68

Úvod

Výučba odboru aplikovanej informatiky na našej fakulte pri veľkom množstve študentov predstavuje pre vyučujúcich množstvo zodpovedností. Je preto dobré, aby vyučujúci mali k dispozícii praktické nástroje pre urýchlenie práce a zlepšenie kvality výučby. V súčasnosti pre uľahčenie ich práce sa používa odovzdávací systém LIST, ktorý bol vyvinutý ako bakalárska [16] a diplomová práca v roku 2013. Opodstatnenie systému je udržiavanie a správa veľkého množstva zadaní úloh, podpora pre odovzdávanie riešení študentov a automatické vyhodnotenie riešení testovaním so sadou jednotkových testov. S odstupom času sa technológie a štýly spracovania aplikácii drasticky zmenili s čím LIST nevedel udržať tempo. Modernizácia LIST-u na najnovšie verzie technológií je drahá záležitosť, čo viedlo k opodstatneniu tvorby úplne nového systému od základov.

Cieľom bakalárskej práce je práve vyhotovenie návrhu pre finálnu štruktúru systému a vytvorenie pevných základov pre dlhodobý vývoj ďalšími študentmi. Podrobne sme preskúmali moderné technológie, existujúce implementácie LIST-u a spolupracovali s jeho hlavnými používateľmi, aby sme vyhotovili jasnú definíciu a návrh systému, ktorý spĺňa všetky potrebné funkcionality a požiadavky. Implementácia bude rozsiahla, pričom sme zohľadnili vhodné návrhové vzory. Finálny systém bude modulárna a ľahko škálovateľná aplikácia, s čím vieme v budúcnosti osloviť všetkých učiteľov na fakulte, aby sa začal využívať jeden systém pre výučbu. Návrh aj implementácie sú taktiež priamou ukážkou zapojenia veľkého množstva technológií, ktoré spolu fungujú ako vyladená symfónia a tvoria robustný systém.

Bakalársku prácu sme rozdelili do troch kapitol. V prvej kapitole uvádzame definície východísk projektu, podrobnú analýzu existujúcich riešení a popísanie technológii, ktoré boli využité vo vývoji projektu.

V druhej kapitole sa venujeme podrobnej analýze požiadaviek od vyučujúcich, z ktorých sa následne odvíja proces návrhu. Uvádzame mnoho iterácii návrhov pre architektúru, model systému a infraštruktúru kde vysvetľujeme opodstatnenia zmien a myšlienok, ktoré nás viedli ku finálnemu návrhu.

Tretia kapitola uvádza realizáciu a implementáciu kľúčových častí systému s ukážkami riešení a popismi postupov.

Kapitola 1

Východiská práce

V tejto kapitole opisujeme východiskové informácie rozdelené do troch častí: teóreticke pojmy, uvedenie a analýza existujúcich systémov a technológie vhodné pre vývoj cieľového systému.

1.1 Teoretické východiská

1.1.1 Learning management system

Learning Management System známy pod skratkou LMS, je softvérová aplikácia alebo webová platforma navrhnutá pre poskytnutie funkcionality spravovania, doručenia potrebného obsahu a sledovanie chodu vzdelávacích kurzov [19]. Systém slúži ako centrálny register a miesto pre administráciu všetkých potrebných aspektov vzdelávania. LMS tvorí kópiu odrazu vzdelávania v skutočnosti nakoľko sa skladá z kurzov, študentov, obsahu výučby, hodnotenia, správy a ďalšie. Základná funkcionalita a charakteristika typického LMS je:

- správa a registrovanie kurzov,
- správa, registrovanie a dodávanie študijného obsahu,
- správa používateľov,
- správa systému, zabezpečenie a integrácie pomocou administratívnych nástrojov,
- podpora mobilného prístupu,
- tvorba úloh, testov a iné formy hodnotenia,
- odovzdanie a hodnotenie riešení študentov,
- sledovanie pokrokov a hodnotení študentov,

- komunikačný kanál.

Hlavná problematika riešená pomocou LMS je zjednodušenie a zefektívnenie štruktúrovania výučby, doručenia obsahu študentom, sledovanie pokrokov a hodnotení na verejnom a zabezpečenom centrálnom mieste.

1.1.2 Domain-Driven Design

Doména v oblasti softvérového vývoju predstavuje oblasť znalosti z ktorej sa vyvodzuje aplikačná logika. Doménová logika aplikácie určuje usmernenia a množinu pravidiel ako sa má objekt domény správať a interagovať s inými objektami, aby sa modelovali potrebné dáta a správania. Doména softvéru je vždy odrazom skutočnej biznis domény [8], napríklad v prípade domény banky sa zobrazujú na softvérovej doméne a jej implementácii všetci účastníci ako banka, zamestnanci, klienti, účty a ďalšie.

Domain-Driven Design je metodika vývoju softvéru, ktorý kladie dôraz a dôležitosť na pochopenie domény vyvíjaného softvéru. Jej cieľom je izolovanie poznatkov domény v častiach, ktoré majú technologickú zodpovednosť namiesto riadenia doménovej logiky, a tvorí tak udržateľný, ľahko škálovateľný a flexibilný celok.

1.1.3 Modul a modularita

Systémy vypracované ako jedna spustiteľná aplikácia a pozostávajúca z jedného alebo viacero úzko spätých a previazaných projektov nazývame monolit [20]. Monolitná aplikácia čelí značnému problému pri údržbe kódu, možnosti rozšírenia a hlavne vyžadujú od vývojára znalosť celej domény, všetkých využitých technológií a samotnej implementácie. Preto ako riešenie pre elementárny problém s monolitmi vznikol princíp modulov a spracovania aplikácii modulárne.

Modul je samostatná časť systému, ktorá je dôležitou časťou väčšieho celku a zapuzdruje súvisiace funkcionality do jedného balíka. Pri moduloch je dôležité, aby sa vzájomne funkcionálne nepokrývali, teda systém pri navrhovaní je potrebné rozdeliť do menších, udržateľných a prepoužiteľných komponentov. Modul vo výsledku nemusí tvoriť spustiteľnú aplikáciu ale aj napríklad knižnicu. Práve v modulárnom návrhu systému je značná výhoda v možnosti ľahkej zámeny za nový modul bez zásahu do kódu druhých modulov so závislosťou na ňu [14], stačí iba zmeniť definíciu závislosti ako je cesta, názov alebo verzia. Samozrejme je predpokladom, že nový modul obsahuje funkcionalitu využitú v moduloch so závislosťou, avšak implementácia metód sa môže vnútorne zmeniť pokiaľ sa zachová rovnaká definícia vstupov a výstupov. Taktiež modularita systému umožňuje prácu viacero pracovných buniek na veľkom systéme, kde bunky si rozdelia moduly. Vývojári budú tak odľahčení o nutnosť mať vedomosti o celej doméne a celého systému.

1.1.4 Hexagonálna architektúra

Hexagonálna architektúra, často známa aj pod názvami Ports and Adapters architecture a Onion architecture [29], je softvérový architektonický vzor s cieľom tvorby vysoko modulárneho systému. Princíp architektúry je oddelenie jadra systému od vonkajších závislosti ako frameworky, komunikácia s databázou a používateľské rozhranie. Doména a jej logika predstavuje jadro hexagonálnej architektúry a je obklopená sériou vrstiev alebo adaptérov. Adaptér slúži ako brána medzi jadrom a externými komponentami. Ako základ hexagonálnej architektúry definujeme tri hlavné komponenty:

- Jadro systému: kompletný doménový model obsahujúci pravidlá, usmernenia a implementácie logiky a funkcionality domény.
- Porty: rozhrania definované jadrom, ktoré zavádzajú interakcie potrebne s vonkajším svetom. Medzi portmi patria rozhrania pre perzistenciu, správy alebo používateľské rozhranie.
- Adaptéry: implementácie portov, ktoré spájajú jadro s externými komponentami. Adaptéry spracovávajú prekladanie dát medzi jadrom a vonkajšími systémami.

S hexagonálnou architektúrou v systéme zavedieme tri hlavné výhody. Oddelenie zodpovednosti a záujmov, flexibilitu a prenositeľnosť [37]. Oddelením zodpovednosti sa systém stane ľahko prehľadný, jadro je ľahko testovateľné a udržiavateľné bez potreby závislosti na komunikáciu s databázou a zapojenia technologických vymoženosti. Adaptéry sú ako moduly ľahko a flexibilne nahraditeľné novou implementáciou bez vplyvu na jadro, čo umožňuje flexibilný vývoj pri nových požiadavkách alebo modernizácia technológii. S osamostatnením jadra má systém výhodu pri migráciách a prepoužitie v iných kontextoch. Návrhový vzor je čistý a modulárny, uľahčuje pochopenie, údržbu a rozšírenie systému.

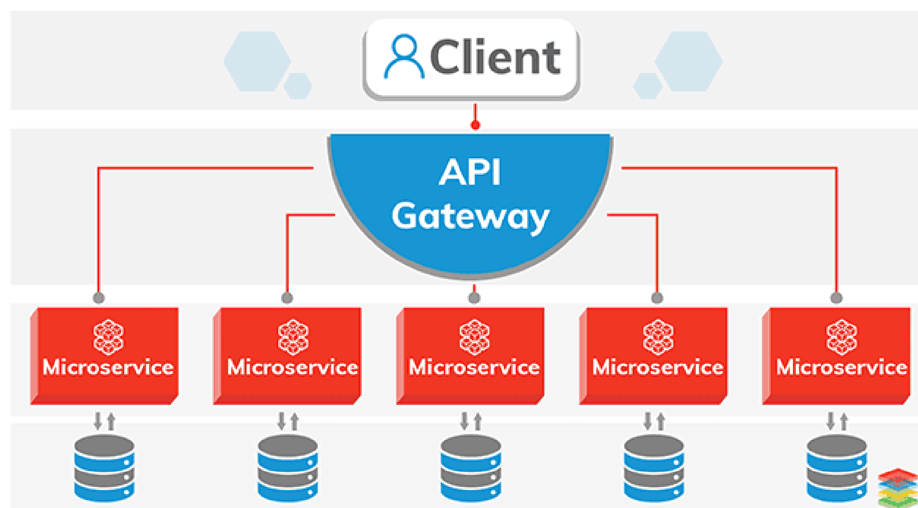
1.1.5 Mikroservisná architektúra

Mikroslužby alebo mikroservisná architektúra je softvérový architektonický vzor a princíp vývoja softvéru tak, aby sa skladal z kolekcie malých a nezávislých služieb. Cieľová aplikácia je rozdelená do viacero samostatných menších aplikácii, s určenou zodpovednosťou a podporou pre unikátne doménové funkcionality. Služby majú jasne stanovené hranice funkcionality, ktoré ponúkajú a vzájomne sa neprekrývajú.

V dnešnej dobe keď sú aplikácie často vyvíjané pre cloud-ove prostredie je mikroservisná architektúra najčastejším výberom pri návrhu systémov. Decentralizovaním zodpovednosti služieb [7] vieme ľahko priradiť podmnožinu služieb mnohým teamom, kde nie je nutné, aby poznali doménovú logiku ostatných služieb, čím sa zvýši efektivita

práce pri vývoji a umožňuje použitie rôznych technológii medzi službami a implementáciu podľa potrieb jednotlivej služby. Veľkou výhodou je odolnosť voči výpadkom, kde zlyhanie jednej služby nevedie k výpadku celého systému.

Vzájomne nezávisle služby z hľadiska domény, jednotlivé služby majú vlastnú podmnožinu domény, umožňuje rozdelenie a rozloženie záťaže databázy. Klient odosielajúci dopyt na systém v skutočnosti posiela dopyt na vstupnú bránu systému, ktorý spracuje dopyt a podľa potreby presmeruje a pošle dopyty jednotlivým službám a následne spracuje odpovede a odosiela klientovi spracované hromadné dopyty. Presné znázornenie stavby systému s mikroservisnou architektúrou vidíme v obrázku 1.1. V častom prípade práve rozdelenie na jednotlivé mikroslužby a záťaže systému vedie k zlepšeniu výkonu a rýchlejším odpovediam.



Obr. 1.1: Skladba mikroservisnej architektúry. [22]

Použitím mikroslužieb vie aplikácia dynamicky škálovať, kde pri nadmernej záťaži služby je ľahké vytvoriť druhú inštanciu tej istej služby a následne rozložiť záťaž medzi nimi. Pre podporu uvedeného dynamického škálovania je potrebná kontainerizácia služieb pomocou Azure Kubernetes, analyzovaný v kapitole 1.3.6 alebo register služieb ako Consul, popísaný v kapitole 1.3.5.

1.1.6 Unit testing

Unit testing preložené ako jednotkové testovanie je v programovaní metodou testovania a verifikovania softvéru ako celok a jeho jednotlivých jednotiek kódu [10]. Jednotka kódu sa rozumie ako najmenšia časť testovateľného kódu, ktorá vie byť nezávisle testovaná. Hlavným cieľom jednotkového testovania je zabezpečenie funkcionality a integrity časti existujúceho kódu, ako triedy či metódy, čím zaručujeme správnosť jeho implementácie.

Testy obvykle tvoria programátori ako časť vývoja a mali by byť automaticky spúšťané počas fázy vývoja a údržby. Programátor ich využíva ako usmernenie a kontrola pri vývoji novej časti alebo pri zásahu do už existujúceho celku. Testy sú izolované od závislosti a iných časti, čo dosiahneme mock objektami, testovacími dátami či inými umelými implementáciami nahrádzajúcimi potrebné závislosti. Izoláciou dosiahneme zjednodušené pátranie po chybách, nakoľko máme plnú kontrolu nad jednotkou bez vonkajšieho vplyvu.

Unit test rozdeľujeme na množinu test case-ov, kde každý test case testuje unikátny prípad v rôznom stave aplikácie. Členením na prípady overujeme správnosť systému za každého predpokladu a do veľkej miery pomáhajú k odhaleniu miesta chyby v kóde. V rôznych metodikách programovania ako Extrémne programovanie a test-driven development (TDD) sa považujú práve unit testy za najdôležitejšiu časť vývoja, kde unit testy sú vytvorené skôr ako sa začne vývoj implementácie.

1.2 Existujúce systémy

1.2.1 Moodle

Moodle je jeden z najznámejších LMS systémov spracovaný ako webová aplikácia. Ako open-source produkt vyvíjaný od roku 2002 jeho najpodstatnejšími výhodami je modularita, moderné spracovanie, aktívna podpora a možnosť využiť verejné a vlastné pluginy. Moodle ponúka široký výber funkcionalít [23] a nástrojov zohľadňujúce potreby vzdelávacích inštitúcii, vyučujúcich a študentov. Systém je flexibilný a konfigurovateľný, čím umožňuje prispôsobiť vzhľad, zapracovať ho v rámci vlastného systému, spravovať kurzy, obsah pre výučbu, hodnotenie pokrokov študentov, spoluprácu medzi študentmi a komunikovať prostredníctvom kanálov na štýl fóra alebo správ.

Vyučujúci majú možnosť nahrávať a zverejniť obsah ako dokumenty, média a kvízy, čím vie štruktúrovať celý priebeh kurzu s prednáškami a hodnotiacim systémom. Hodnotiaci systém sa skladá z kvízov, zadaní a prehľad úspešnosti. Moodle obsahuje možnosť vytvorenia úlohy s automatickým vyhodnocovaním, čím uľahčuje prácu vyučujúcim. Taktiež pre zadania typu programátorských úloh je podpora pomocou pluginu automatický vyhodnotiť správnosť riešenia spustením jednotkových testov. Pre Moodle v dnešnej dobe existuje množstvo pluginov vrátane pluginov podporujúce spúštanie a testovanie kódov. Pluginy žiaľ neponúkajú dostatočnu kompatibilitu a možnosť konfigurácie pre využitie na našej fakúlte. Veľkou výhodou je možnosť ho využiť ako nástroj pre *Massive open online courses (MOOC)*, virtuálnu výučbu, vďaka ohľadu na bezpečnosť, podpore mnohojazyčnosti stránky a možnosti škálovania a obsluhy tisíce používateľov súčasne.

1.2.2 Long-term Internet Storage of Tasks

Long-term Internet Storage of Tasks známy pod skratkou LIST je LMS webová aplikácia, ktorá vznikla ako bakalárska práca [16] študenta aplikovanej informatiky Andreja Jursu v roku 2013. Jej primárnym cieľom je podpora manažovania a dlhodobého ukladania úloh pre výučbu informatických predmetov. Doposiaľ aktívne využívaný ako primárny odovzdávaci a hodnotiaci systém programátorských úloh pre kurzy v odbore aplikovanej informatiky. V priebehu 10-ich rokov jeho existencie bolo vytvorené vyše 3000 úloh, 5000 zostav úloh a presne 140 kurzov. LIST je rozsiahly systém, ktorý podporuje a spĺňa charakteristiku LMS:

- správu používateľov, kurzov a úloh,
- realizácia úloh a cvičení,
- sprostredkovanie učebného materiálu,
- automatické vyhodnocovanie úspešnosti pomocou jednotkových testov,
- manuálne vyhodnotenie zadaní,
- priebežný prehľad úspešnosti študentov,
- zálohovanie súborov a databázy,
- zaznamenávanie činnosti používateľov,
- využitie služby kontroly plagiátorstva systému MOSS od Stanfordskej univerzity.

Taktiež sú zakomponované ďalšie rozsiahle funkcie ako prepínanie medzi jazykmi pre statické aj dynamické texty, skúškový mód pre zakázanie sťahovania riešení a pripojenie z nefakultnej IP adresy počas skúšky (žiaľ nie je implementované pre kurzy zvlášť, ale v prípade skúšky pre jeden kurz tak celý systém LIST je uzamknutý a zablokovanie obmedzí všetky kurzy) a dynamické prepočítanie bodov študenta podľa vzorcov.

Vďaka pokročilosti technológii spätne v roku 2013, LIST bol vyvinutý pomocou PHP a mnoho technológií tretích strán:

- DataMapper ORM pre mapovanie PHP objektov na tabuľky a stĺpce v databáze,
- Smarty 3 templatovací engine pre dynamické generovanie HTML obsahu,
- Codeigniter framework: zbierka knižníc a mnohých funkcionalít, prepoužitelných komponentov a balík pre podporu webovej bezpečnosti,
- jQuery,

- TinyMCE WYSIWYG *(What you see is what you get)* editor.

Výber technológii v súčasnej dobe je vysoko obmedzujúci [17] v prípade potreby úprav alebo pridanie novej funkcionality programátorom, ktorý nie je autorom LISTu. Pre dosiahnutie potrebného rozšírenia funkcionality je potrebné sa oboznámiť s mnoho technológiami, aktuálny štýl riešenia mapovania objektov na databázu, a nevýhoda interpretovaného jazyka absencia možnosti debugovania spolu s komplikovaným prepojením Smarty template-ov.

Systém a jeho prvky sú plne konfigurovateľné učiteľmi prostredníctvom používateľského rozhrania. Napríklad učitelia pri použití automatických testov musia nakonfigurovať správanie a možnosti testovacích skriptov. Avšak LIST nie je najvhodnejšie navrhnutá, vyučujúci majú občas problémy so svižným zapracovaním zmien. Práve rozsiahla konfigurácia komponentov a systému zaviedla mnoho nepoužitých prepínačov a funkcii, zbytočné zovšeobecnenie a hlavne komplexnosť rozhrania. Ako príklad zobrazený v obrázku1.2 máme formulár pri vytvorení zostavy úloh pozostavajúce zo štyroch tab-ov, v ktorých je zavedených množstvo nadbytočných políčok, neprehľadné spracovanie vzhľadu a vyžaduje komplexnú konfiguráciu autmatických testov.

Obr. 1.2: Konfiguračný formulár v LIST-e.

Ako ďalší príklad nevhodného UI/UX návrhu zobrazený na ukážke v obrázku 1.3 je komplexnosť a nepraktickosť navigácie na učiteľskom rozhraní. Navigácia je počiatočne skrytá a z hľadiská UX sa predpokladá, že pri jeho otvorení sa obsah na pozadí nemení a máme možnosť s ním napriek tomu pracovať. V prípade LISTu navigácia odsúva a zablokuje použitie obsahu na pozadí. Ďalším postrehom je spracovanie obsahu navigácie samotného, kde ak v prípade kliknutí na vnorený zobrazenie zoznamov možností stránok, prvky vnoreného zoznamu sa stáva obsahom celej navigácie. Vnorený zoznam v LISTe častokrát tvorí iba pár možností teda nie je potrebné, aby celú obrazovku prekrývalo menu s dvoma tlačidlami. Optimálnejšie spracovanie by bolo vhodnejšie rozdeliť funkcionality do kategórii, ktoré by pri rozkliknutí rozbalili collapsable menu.

Obr. 1.3: Navigácia v učiteľskom rozhraní LIST-u.

Vidíme, že LIST z hľadiska spracovania používateľského rozhrania obsahuje kolíziu so základným princípom LMS, ktorý by mal byť prehľadný a zaručiť jednoduchú a pohodlnú správu výučby.

Ako úlohy podporuje čisto iba programátorské zadania, kde vyučujúcim chýba možnosť vytvorenia kvízov s automatickým vyhodnotením ako obsahuje Moodle. Používateľské rozhranie sa delí na typy, študentské a učiteľské, ktoré majú rozdielny vzhľad. Ako následok rozdelenia rozhraní sa ukázal značný nedostatok v návrhu systému, kde pri využití študentov ako výpomoc pri výučby a opravovaní riešení v kurzoch pre nižšie ročníky je potrebné im vytvoriť osobitné učiteľské konto, čím dostane plný prístup k systému a rovnaké právomoci ako administrátor. Teda okrem možnosti konfigurácie celého systému, študent má prístup k celému obsahu LISTu vrátane jeho vlastných kurzov.

Funkcia automatického testovania a vyhodnotenia programátorských úloh funguje na princípe shell skriptov, ktorý dostane študentské riešenie a zbierku jednotkových testov, spracuje a sformatuje študentov kód, spustí jednotkové testy a na výstupe vráti úspešnosť riešenia. Pri spustení testu je najprv vytvorený dočasný adresár s náhodne

```
// Mapovanie jedneho objektu studenta
$student = new Student();
$student->get_by_id($this->usermanager->get_student_id());

// Mapovanie zoznamu objektov studentov podla podmienky
$student = new Student();
$student->where_related('participant', 'allowed', 1);
```

Kód 1.1: Ukážka využitia ORM mapovania v LIST.

vygenerovanou cestou pre zamedzenie prístupu k ostatnému systému, v ktorom sa spusti proces testovania. Ako zásadná chyba prístupu testovania je absencia zamedzenia prístupu na internet a prázdny kontajner využitý pre testovanie. V aktuálnom štýle testovania má študent možnosť využiť rôzne nemravné techniky pre preniknutie do systému, avšak LIST podporuje možnosť kontainerizácie cez docker čím sa dá zabezpečiť testovanie prostredie riešení.

LIST je systém, ktorého cieľom malo byť riešenie šité na mieru pre našich vyučujúcich, teda je aj predpokladom že problémy alebo možné rozšírenia budú vykonané vyučujúcimi alebo inými študentmi. LIST ako projekt má značný problém s modularitou a možnosťou pridania nových komponentov. Samotná implementácia domény spolu s mapovaním pre relačnú databázu je mätúca, nakoľko triedy objektov nepredstavujú vždy to isté. Napríklad pre reprezentáciu študentov máme triedu *Student*, ktorá podľa názvu by mala predstavovať jeden objekt. Ako je môžeme vidieť v kóde 1.1 s ukážkou spôsobu mapovania dát na objekty, pre získanie namapovaních inštancii entít je potrebné vytvoriť prázdnu inštanciu študenta, s ktorou vieme následne namapovať objekty [17]. Ako výsledok podľa typu využitej metódy získame jeden namapovaný objekt alebo zoznam objektov, čo znamená že je zavedená nekonzistencia a mnoho mätúcich premenných. Taktiež trieda predstavujúca doménový objekt má funkcionalitu a závislosti, ktoré sú pre ňu nepodstatné. Ako alternatíva by bolo vhodné aby objekt prestavujúca entitu a jej mapovanie bolo oddelené, napríklad na entitnú triedu *Student* a službu *StudentMapper* ktorej zodpovednosťou bude práve vykonanie operácii pre namapovanie objektov typu Student.

1.3 Technológie

1.3.1 Object-Relational Mapping

Object-Relational Mapping (ORM) je programovací nástroj ktorý pomocou deskriptora metadát transparentne prepojí objekty a triedy s relačnou databázou [12]. ORM zabezpečuje automatické mapovanie databázových dát na objekty a objekty na dáta v databáze. Existuje mnoho nástrojov implementujúcich mapovanie entít kde v Java najznámejšie sú TopLink, EclipseLink alebo Hibernate, ktorý popisujeme v kapitole 1.3.2.

ORM poskytuje abstrakciu nad relačnou databázou a jeho využitím sa zbavíme potreby udržovania SQL dopytov nad databázou a manuálneho spracovania výsledkov. Pre jeho použitie je potrebné zadefinovať vzťahy jednotlivých entít a ich atribútov s tabuľkami a stĺpcami v databáze. V Java sa tieto vzťahy častokrát definujú vrámci tried pomocou anotácií alebo v XML súboroch podľa upresnení špecifického ORM nástroja. Nástroj primárne zvyšuje produktivitu vývojárov, znižuje opakujúci sa kód a pomáha pri jej údržbe.

1.3.2 Hibernate ORM

Hibernate ORM zjednodušene Hibernate je open-source ORM nástroj pre Java aplikácie. Hibernate ako framework implementuje primárnu funkcionalitu ORM, zjedodušenie vyhľadávania, validáciu doménového modelu a narábanie s jej entitami na perzistenčnej úrovni [13]. Hibernate spravuje životný cyklus objektov a ich stavy v relačnej databáze, teda okrem tradičných CRUD operácii *(Create, Read, Update, Delete)* spracováva objektové vzťahy, asociácie, lazy-loading, cache-ovanie a riadi databázové transakcie.

Hibernate umožňuje funkcionalitu nad rôznymi databázami pomocou abstraktnej vrstvy, umožňujúcej jednoduchú zmenu typu databázy. Pre vývojárov zahŕňa asbtrakciu nad databázovým modelom s ponukou vlastného dopytového jazyka Hibernate Query Language (HQL) [34]. HQL je totožný jazyk s SQL pri čom rozdiel je tvorba dopytov nad Java objektami namiesto databázových tabuliek. Vývojári v dopytoch využívajú názvy tried a ich atribútov, čím ich odľahčuje o znalosť databázového modelu a zároveň domény Java tried.

Pre zapojenie nástroja a jeho využitie je potrebná definícia a konfigurácia objektov a vzťahov k databázovým tabuľkám a stĺpcom. Hibernate podporuje definíciu mapovania pomocou mapovacieho XML súboru alebo anotáciami. Základný príklad mapovania entitnej triedy pomocou anotácii je zobrazený v ukážke kódu 1.2. Obidva prístupy zohľadňujú aj komplexné mapovacie potreby ako vnáranie objektov, kolekcie a relácie vrátane typu many-many.

```
1   @Entity
2   @Table(name = "user_table")
3   public class User {
4
5       @Id
6       private Long id;
7
8       @Column
9       private String name;
10  }
```

Kód 1.2: Ukážka Hibernate anotácií.

1.3.3 REpresentational State Transfer

REpresentational State Transfer alebo REST, predstavuje architektonický návrh pre webové služby distribuované a sieťové aplikácie [9]. Myšlienkou REST je definovanie normiem pre komunikáciu prostredníctvom HTTP protokolu, štruktúrovanie obsahu dopytov a odpovedí a jej zapracovanie do aplikácií. Kladie sa dôraz na škálovateľnosť a koordinovanosť komunikácie medzi systémami. Službu môžeme označiť ako RESTful službu iba v prípade, ak dodržiava základné vlastnosti:

- Stateless communication: komunikácia klienta so systémom je bezstavová, teda každý dopyt obsahuje potrebný obsah pre spracovanie požiadavky a server si neukladá žiadne informácie o klientovi.
- Jednotné rozhranie: RESTful služba je vymedzená rozhraním, definujúca každý zdroj využitý pri komunikácii klienta so serverom.
- Klient a Server: rozdelením klienta a servera zaručujeme možnosť škálovania a osobitný vývoj. Ako jediné je potrebné dohliadnúť aby rozhranie medzi klientom a serverom bolo jednotné a neporušené.
- Reprezentácia a typy médií: RESTful služba s klientom majú dohodu pre podporované typy médií ako HTML, JSON alebo XML.
- Vrstvy služby: Služba sa skladá z vrstiev a komponentov s obmedzenym správaním.

Teda RESTful služba prijíma dopyty na definovaných cestách a presne určenými atribútmi v ceste alebo ako payload, s definovanou metódou dopytu. RESTful služby sú modernou technológiou, ktorá je často využívaná na jednoduché zavedenie komunikácie medzi službami cez HTTP protokol.

1.3.4 Spring

Spring Framework je open-source aplikačný framework, ktorý ponúka komplexnú a rozsiahlu podporu pre tvorbu podnikových (enterprise) Java aplikácii [24]. Kľúčovým cieľom je zvýšenie produktivity a efektivity práce vývojárov dosiahnutý sprostredkovaním jeho robustnej infraštruktúry, širokým výberom prepoužiteľných komponentov a neustálym vývojom svojich knižníc.

Základnou charakteristikou Springu je jej podpora pre princípy:

- Inversion of Control (IoT): neexistuje objekt s kontrolou ako typická main metóda, ktorá volá a vykonáva sekvenčne príkazy ale namiesto toho objekty sú inštancovné pri zapnutí služby a registrované ako Java Bean čakajúce na volanie jej metódy,
- Dependency Injection: V prípade závislosti jedného Java Bean na iný Java Bean objekt, nevytvára sa ďalšia inštancia potrebného objektu, ale adresuje a využíva sa jej už existujúca inštancia.

Okrem podpory pre uvedené princípy, primárne využitie Spring framework-u je využitie ORM nástrojov, REST-ových API pre webové aplikácie, jednoduché možnosti integrácii s rozlišnými technológiami a framework-mi [36] ako asynchronné správy (JMS), webové služby, podpora práce s inými knižnicami a mnoho ďalších. Základ Spring-u sa skladá z 20 modulov pokrývajúce potreby pre komunikáciu prostredníctvom internetu, prístup k dátam a ich integrácia, kontajnerizácia služieb a jej komponentov, messaging, aspect-oriented programovanie, testovanie a široká infraštruktúra pre kompatibilitu a jednoduché zapojenie s inými technológiami. Existuje veľké množtvo Spring knižníc, z ktorých každá jedna je vybudovaná práve z týchto core modulov. Vďaka tejto robustnej infraštruktúre a širokej dostupnosti funkcionality umožňuje vývojárom ľahko tvoriť škálovateľné, modulárne, udržateľne a robustne systémy.

1.3.5 Consul

Hashicorp Consul, tiež ako iba Consul, je open-source softvér, ktorý slúži ako centralizovaný register služieb na rozlíšených prostrediach [11]. Služby na prostrediach so správnou konfiguráciou sú pri naštartovaní automaticky registrované v consul, ktorý monitoruje zdravie služieb a ich dostupnosti. Consul pracuje na princípe master a worker nodes, kde každý worker node je na inom prostredí, vykonáva potrebné monitorovania, registrácie a dopyty, ktoré následne sú odoslané master node na spracovanie. Consul neobsahuje žiadne automatizácie ani inteligentné orchestrácie, ale najmä slúži ako light-weight riešenie na monitorovanie iných službieb.

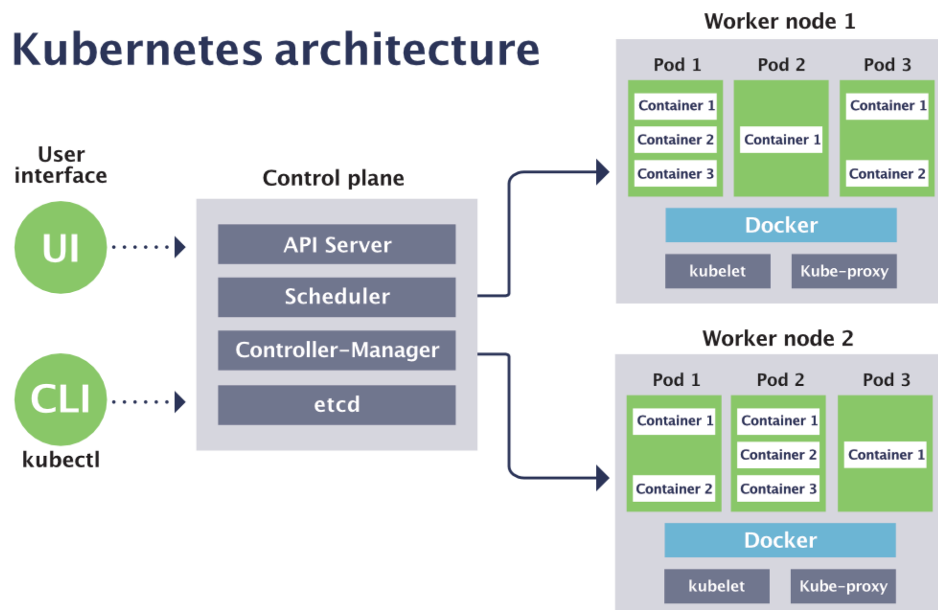
V prípade viacero inštancii jednej služby je load-balancing riešený klientom, teda on si určuje ako často si žiada adresu inštancií a ktorú následne použije. Pri zlyhaní

kontroly dostupnosti niektorej z registrovanej inštancii, consul automatický podáva informáciu potrebným službám o jeho výpadku, aby sa zabránilo zlyhaným dopytom a možným problémom.

Okrem registrovania a udržovania stavov služieb, consul slúži aj ako centralizované úložisko aplikačných properties. Properties sú odoslané službám pri ich počiatočnom registrovaní do consul, čím vieme tieto properties modifikovať bez potreby buildovania artefaktov.

1.3.6 Azure Kubernetes

Azure Kubernetes je služba, ktorá obsahuje totožnú funkčnosť ako Consul a navyše rieši rôzne automatizácie, orchestrácie a zjednodušenia škálovateľnosti. Azure Kubernetes je spoľahlivá a škálovateľná služba od Microsoft na automatickú a manažovanú orchestráciu kontajnerov [21]. Kontajner je izolované prostredie pre aplikácie či služby ako virtuálny počítač, po anglický virtual machine (VM), bez virtualizácie fyzického hardware-u počítača. Architektúra systému s Kubernetes pozostáva zo skupiny virtuálnych strojov (VM) označovaných ako nodes, kde jeden slúži ako master node či control plane a ostatné nodes sú jemu podradení workeri. Podobne ako v mikroservisnej architektúre s API gateway, master node slúži ako gateway ku worker nodes, čo je zobrazené na obrázku 1.4. Master node okrem distribúcie dopytov a load-balancing má na starosti scheduling, manažovanie celého clustra, monitorovanie worker nodes a automatické škálovanie systému.



Obr. 1.4: Ukážka architektúry Kubernetes cluster. [31]

Škálovanie kubernetes systému je spracované ako pri mikroservisnej architektúre, teda je možnosť mať mnoho inštancii služby na akomkoľvek prostredí. Škálovanie je dynamické vďaka aktívnemu monitorovaniu podľa čoho master node rozhoduje o založení a registrácii nového worker node s potrebnými kontajnermi a službami, alebo môže už v existujúcom worker node pridať novú inštanciu služby. Vďaka takejto automatickej orchestrácii, systémy s Kubernetes majú tiež výhodu s použitím nástrojov pre Continuous Integration and Deployment (CI/CD) čím vedia dosiahnúť bezvýpadkove nasadenie aplikácií. Využíva sa tu podpora automatického škálovania, vďaka čomu sa vytvorí nový worker node s novými artefaktmi, ktorý následne po úspešnom nasadení node a artefaktov nahradí node so starou verziou artefaktov. Prínosom využitia Azure Kubernetes je umožniť teamom venovať sa vývoju a manažovaniu aplikácii bez starostí s infraštruktúrou.

1.3.7 Feign

Feign je deklaratívny REST klient pre webové služby poskytnutý v rámci projektu Spring Cloud Netflix. Pomocou rozhrania anotáciou feign klienta a správnou konfiguráciou, umožňujeme aplikácii podávať dopyty prostredníctvom HTTP protokolu na odľahlú RESTful službu a prijímať odpovede [35].

Rozhranie pre feign klienta musí obsahovať definíciu koncových bodov, metóda dopytu *(GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS)*, hlavičky a očakávaný obsah tela. Pre zabránenie duplicitnému kódu a nutnosti udržovania definícií RESTful API na viacerých miestach, teda API samotné a všetky rozhrania feign klientov pre dané API, Spring umožňuje využiť takzvané *boilerplate interfaces* kde pomocou dedičnosti vieme zdediť definíciu priamo od zverejneného rozhrania API definície. Pri použití viac inštancovaných služieb v architektúre systému, v Spring ekosystéme existuje nadstavba feign klienta s podporou pre automatické požiadanie o adresy danej služby z registru inštancií ako je consul a následný load balancing.

1.3.8 Angular

Angular je známy open-source framework pre vývoj single-page webových aplikácií na enterprise úrovni od spoločnosti Google. Aplikácia sa skladá z typescriptových kódov s HTML predlohami [1]. V Angular je zaužívaný princíp tvorby znovupoužiteľných komponentov, komunikácia a interakcia medzi použitými komponentmi. Uplatňuje sa Model-View-Controller (MVC) architektúra, čo poskytuje štrukturovaný prístup k správe aplikačnej logiky, komponenty používateľského rozhrania a data-binding z aplikačnej logiky na grafickú časť používateľa.

Angular knižnice zahŕňajú široké množstvo funkcionality pre typescriptové kódy ako interceptor dopytov a odpovedí, služby registrované a dodané na potrebných mies-

tach podobne ako Java Beans a dependency injection v Spring, router ktorý umožňuje komplexné a podmienené navigovanie vrámci aplikácie, validácia formulárov a mnoho daľších. Knižnica RxJs sa častokrát používa spolu s Angular a rieši najmä odosielanie dopytov so spracovaním odpovedí a dynamickú transformáciu dát po zmenách stavu komponentov alebo aplikácie.

1.3.9 Bootstrap

Bootstrap je open-source frontend framework pre vývoj responzívnych webových aplikácii. Obsah frameworku sa skladá z znovupoužiteľných jednotiek s designom pre používateľské rozhranie, dynamické a responzívne rozmiestnenie prvkov v grid systéme a iné JavaScriptové plugin-y. Bootstrap zaručuje kompatibilitu pre široký výber prehliadačov.

Angular ponúka vlastnú nadstavbu Bootstrapu s názvom ngBootstrap [1], kde okrem tradičného Bootstrapu sú zahrnuté znovupoužíteľné komponenty s podporou prispôsobenia podľa vlastných požiadaviek a rozšírené funkcionality štandardnej Angular knižnice.

1.3.10 Angular Material

Angular Material je open-source framework na podobný princíp ako ngBootstrap, kde sú ale dostupné komplexné komponenty, zahŕňajú typescriptovú logiku spolu s designom a animáciami. [2] Framework obsahuje znovupoužiteľné komponenty ako sú:

- tlačidlá a skupina tlačidiel s animáciami a ich správanie,
- kontainerizované kartičky so štruktúrovaným obsahom,
- rôzne typy políčok pre vstup s extra vymoženosťami, napríklad textový vstup s možnosťou automatického dopĺňania textu podľa dostupných možnosti,
- spracovanie a zobrazenie stromovej hierarchie objektov s možnosťou dynamického rozbaľovania potomkov,
- paginátor, ktorý zahŕňa aj registrovanie a spracovanie zmien od používateľa,
- dynamické modálne okná,
- flexibilná mriežková štruktúra s responzívnym triedením prvkov,
- rôzne komponenty pre zobrazenie správ ako overlay,
- tabuľka, ktorej stačí definovať predlohu riadkov a zdroj dát a postará sa o dynamické načítanie dát, zobrazenie a jeho štruktúrovanie,

- navigácia a rôzne menu,
- tooltipy.

Angular Material je naozaj bohatá a postačujúca nadstavba ngBootstrap pre interaktívne komponenty. Hlavnou súčasťou je aj podpora vytvorenia vlastnej témy s troma farebnými paletami, ktoré sú použité vo vzhľadových štýloch použitých Angular Material komponentov.

1.3.11 Thymeleaf

Thymeleaf je server-side template-ovací engine založený na Java, ktorý sa používa na generovanie dynamických HTML stránok [33]. Template-ovací engine Thymeleaf umožňuje spracovaniu dopytov a vygenerovanie stránok na serveri, ktoré ako finálny produkt su poskytnuté používateľovi. Využíva špeciálne označenia v HTML predlohách pre placeholder, operácie ako podmienky či cyklus. Hlavnou výhodou je skutočne ponechanie celej logiky aplikácie na serveri a nezaťažovať klienta so spracovaním obsahu a funkcionality. Naopak nevýhodou môže byť vývoj samotný, nakoľko Thymeleaf je engine, ktorý generuje stránky za behu aplikácie, pre vývoj a testovanie je potrebné rátať s neustálym refreshom kontextu a zdrojov.

1.3.12 Nginx

Nginx je známy open-source webový server s podporou pre reverse proxy. Je známy pre jeho efektívnosť, výkonnosť a škálovateľnost. Ako webový server je schopný obslúžiť veľké množstvo dopytov, vrátane súbežných dopytov. Nginx slúži tiež ako riešenie pre reverse proxy, čo znamená dopyt klientov je analyzovaný a podľa konfigurácie a potreby preposiela dopyt cieľovej backend službe na potrebnom prostredí a spätne presmeruje odpoveď klientovi. Navyše Nginx ponúka široký výber ľahko zapojiteľných modulov a knižníc z jeho ekosystému [26].

Kapitola 2

Návrh systému

Hlavným cieľom práce je položenie základov pre nový systém nahrádzajúci LIST. Preto je návrhová časť práce najkritickejšia, aby v prípade rozširovania funkcionality nečelil vývojár kolíziám so základmi celého systému. V tejto kapitole uvádzame high-level popis domény, výsledky konzultácií s vyučujúcimi ohľadom nutných požiadaviek s možnými dodatkami a verzie návrhov s popismi a odôvodnením rozhodnutí.

2.1 Požiadavky a očakávania

2.1.1 Popis systému

LIST-NG ako správny LMS systém musí spĺňať základnú charakteristiku LMS. Základné termíny, ktoré vystupujú a tvoria základ v aktuálnom systéme LIST [16][17] sú:

- *Používateľ*: interagujuje so systémom a rozlišuje sa prístup podľa typu (študent, učiteľ)
- *Obdobie*: označuje časové obdobie v ktorom sa konajú kurzy, často interpretované ako semester,
- *Kurz*: vyučovací predmet vedený aspoň jedným učiteľom,
- *Miestnosť*: predstavuje fyzickú miestnosť na fakulte so stanovenou kapacitou,
- *Skupina*: skupina používateľov v kurze, často využité na rozdelenie študentov podľa krúžkov,
- *Vyučovacia hodina*: opakujúce sa vyučovanie na týždennej báze v rovnaký deň, miestnosti a čase pre určenú skupinu v kurze,

- *Úloha*: všeobecný záznam úlohy pre študentov s popisom, súbormi na stiahnutie a unit testami,
- *Zostava úloh*: zbierka vybraných úloh zverejnená študentom na vypracovanie a odovzdanie riešenia s definovaným spôsobom a váhou hodnotenia,
- *Test*: testovací scenár vo forme unit testov pre jednu úlohu v zostave úloh,
- *Riešenie*: študentove odovzdané riešenie pre špecifickú zostavu úloh,
- *Hodnotenie riešenia*: buď automatické, alebo ručne učiteľom zadané hodnotenie študentovho riešenia,
- *Kategória*: označenie typu obsahu a učebnej látky pre úlohu,
- *Typ zostavy úloh*: rozlíšenie žánru zostavy úloh ako cvičenie, domáca úloha, projekt, skúška alebo bonus,
- *Príloha*: súbor, ktorý sa nahráva ako riešenie ku zostave úloh, viditeľný materiál na stiahnutie, doplnok ku textu v úlohe alebo skrytý súbor potrebný pre administráciu.

Uvedené termíny naozaj sú odrazom skutočných aspektov výučby a tvoria základ domény LIST-NG. Je potrebné zohľadniť možné rozšírenia funkcionality, preto je dôležité spracovať návrh domény abstraktne, aby pri potrebe zmien nevzniklo množstvo kolízii a aspektov ktoré treba brať do úvahy.

2.1.2 Požiadavky

Aby LIST-NG naozaj prenikol ako nahrádzajúce riešenie aktuálneho LIST-u, musí obsahovať minimálne totožnú funkcionalitu. Preto ako nevyhnutnou súčasťou analýzy a návrhu boli konzultácie s aktívnymi používateľmi, ukážky používania a vlastné bádanie LIST-u. Výsledok konzultácii a výskumu je zoznam minimálnej funkcionality, ktorú by mal LIST-NG spĺňať:

- ponechať základy LIST-u a jej doménových relácií pre možnosť migrácie existujúcich dát z LIST-u,
- kategorizovanie úloh podľa kategórií,
- kategorizovanie zostáv úloh podľa typu zostavy,
- typy zostáv úloh sú všeobecné a môžu sa priradiť kurzom,
- možnosť posielania hromadných emailov účastníkom kurzu,

- správu kurzu môže mať na starosti viac než jedna osoba, dokonca aj študent zvolený učiteľom v danom kurze,
- organizované a prehľadné používateľské rozhranie,
- efektívne využitie miesta na obrazovke, minimalizovať prázdny priestor aspoň v rámci učiteľského rozhrania,
- lokalizácia textov (slovensky a anglicky),
- možnosť prípravy zostáv vopred bez jeho zverejnenia študentom,
- kopírovanie kurzu, ktoré bude obsahovať aj jeho konfiguráciu, skupiny a zostavy úloh,
- pre jednotlivé úlohy vytvárať unit testy, otestovať nimi riešenie študenta a zobraziť výsledok úspešnosti,
- zobrazenie a aktualizovanie zostávajúceho času pre odovzdanie riešenia,
- jednoduchá konfigurácia testovania pre zostavu úloh,
- určenie priorít testov pre zostavy úloh, napríklad skúškové zadanie bude mať vyššiu prioritu pre spustenie testov než domáca úloha,
- spustenie testov pre jednotlivé úlohy v zostave, namiesto spúšťania testov pre celú zostavu,
- skúškový mód: zablokovanie sťahovania materiálov a riešení v kurze a možnosť zadania povolených IP adries pre odovzdanie,
- v tabuľkách je potrebná možnosť zoradenia podľa stĺpcov a filtrovania dát,
- zaznamenávanie činnosti používateľov, minimálne sťahovanie a nahrávanie,
- porovnávanie riešení študentov pomocou služby MOSS,
- manuál pre správcu a učiteľov.

Nakoľko LIST vznikol v roku 2013, používatelia mali dostatok času prebádať všetky jeho časti a oboznámiť sa s jeho nedostatkami alebo možné rozšírenia pre zlepšenie kvality. Spolu s učiteľmi aj pomocou vlastných zistení LIST-NG má niekoľko požiadaviek nad rámec jeho základnej funkcionality:

- možnosť vytvorenia zadania vo forme kvízu,
- možnosť vygenerovania statického odkazu pre náhľad úloh a zostavy úloh,

- zapamätanie nastavení a kontextu používateľa (filter, otvorený predmet, ...),
- programovací editor priamo v zostave úlohy. Po odovzdaní riešenia sa súbory načítajú v editore alebo môžeme vytvárať nové súbory priamo v rozhraní,
- viditeľné a tajné testy, kde viditeľné dávajú študentom hodnotenie a tajné testy slúžia ako kontrola pre učiteľov alebo iné potreby,
- testovanie vrámci kontajnerov, čím naozaj dosiahneme izolované prostredie pre testovanie nad ktorou máme plnú kontrolu (konfigurácia verzií SDK a povolených knižníc, zabezpečenie proti manipulácii so systémom, ...),
- podpora vlastného testovacieho formátu, napríklad v prípade úlohy pre kurz programovania v Python postačuje kontrola rovnosti výstupu metódy s očakávanou hodnotou,
- vyhnúť sa zbytočnému rozdeleniu obsahu na viacero okien, napríklad pri vytvorení a editácii zostavy úloh nie je potrebné mať správu textov, súborov a konfigurácie testov na rozličných oknách.

2.2 Analýza

V tejto časti kapitoly sú objasnené verzie návrhov a rozhodnutí jednotlivých časti architektúry. Iterácie návrhu boli vždy prezentované a odkonzultované so školiteľom, ktoré viedli postupne k finálnemu návrhu.

2.2.1 Technológie

Technologické závislosti pre služby a používateľské rozhranie sú najpodstatnejšou časťou rozhodnutí pri návrhu, nakoľko aj malá zmena môže ovplyvniť plán a postupnosť implementácie. Ako základ infraštruktúry by bolo optimálne využiť Azure Kubernetes a mať do budúcna pripravený pevný základ pre škálovanie a automatizácie. Pri hlbšej analýze a zoznámení sa s Kubernetes, sme došli k záveru, že Kubernetes je naozaj technológia pre veľké systémy s potrebou zvládnuť veľkú premávku bez výpadkov. Implementácia infraštruktúry s Kubernetes by bola obtiažna a zdĺhavá. Ako jednoduchšia náhrada implementácie škálovania a podpora mnoho inštancií služieb sme sa rozhodli použiť Consul.

V súčasnej dobe s aktuálnymi pokrokmi v technológiách framework Spring pre vývoj modulov ako služby bol najlepší výber. Spring je naozaj populárnym a obľúbeným framework-om pre vývoj mikroslužieb so širokou podporou integrácií s inými framework-mi. Ako technologické závislosti pre vývoj mikroslužieb v LIST-NG sa zvolil

Spring framework ako základ, Hibernate ako využitý ORM nástroj a Feign pre komunikáciu medzi službami cez HTTP protokol pomocou REST-ových volaní.

2.2.2 Modularita

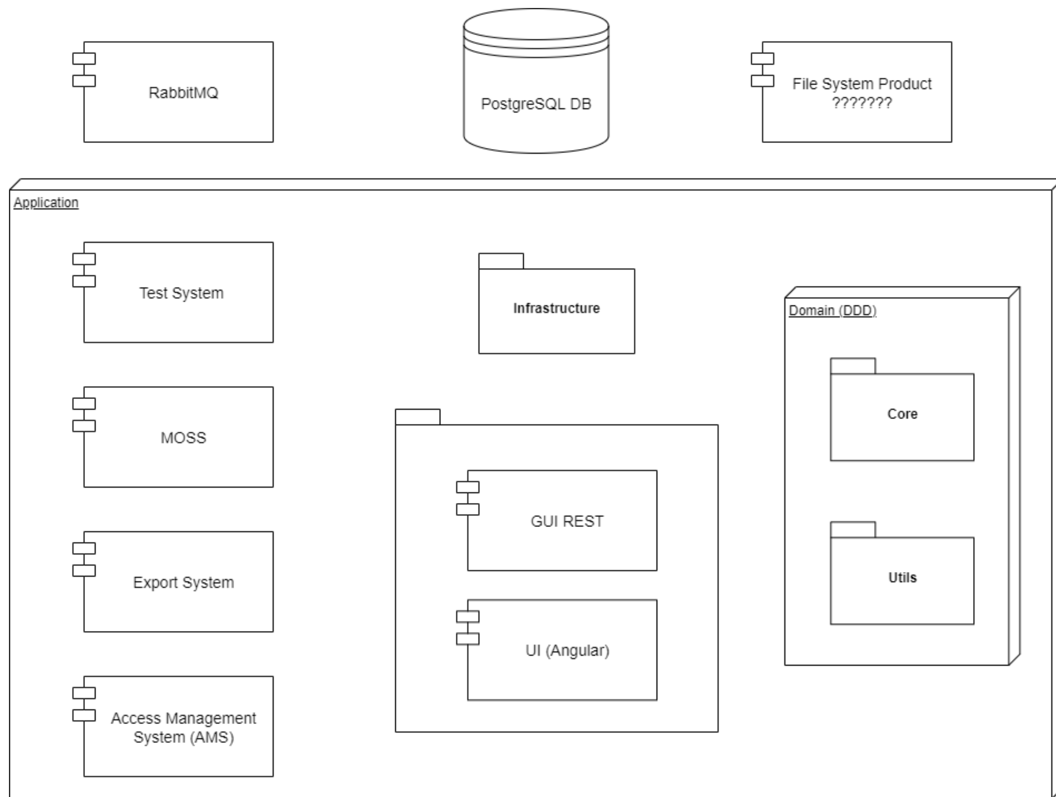
Spracovanie návrhu modularity systému bolo náročné a vyžadovalo si prehľad v doméne systému, podľa ktorého sa bude odvíjať rozdelenie na moduly. Vo fáze analýzy a navrhovania vznikli tri iterácie návrhu pre moduly a ich funkcionalitu.

Prvá iterácia

Na obrázku 2.1 vidíme náčrtnutý prvý návrh modularity. Ako základ systému tvoria:

- *Access Management System*: modul pre správu používateľov, období a kurzov s ich všetkými aspektmi (skupiny, vyučovacie hodiny, miestnosti),
- *Test System*: modul pre správu úloh, zostav úloh a automatické testovanie,
- *Export System*: modul pre vygenerovanie exportu úloh do PDF,
- *MOSS*: modul pre krížové porovnanie riešení študentov a nachádzanie zhôd,
- *GUI REST*: API gateway pre používateľské rozhranie,
- *GUI UI*: Angular webová aplikácia,
- *Infrastructure*: zdieľaná knižnica pre moduly, ktorý bude obsahovať zdieľané implementácie, konfigurácie a definíciu závislosti a ich verzii,
- *Core*: projekt s doménovým jadrom systému,
- *Utils*: knižnica pre projekt Core s rozšírujúcou funkcionalitou.

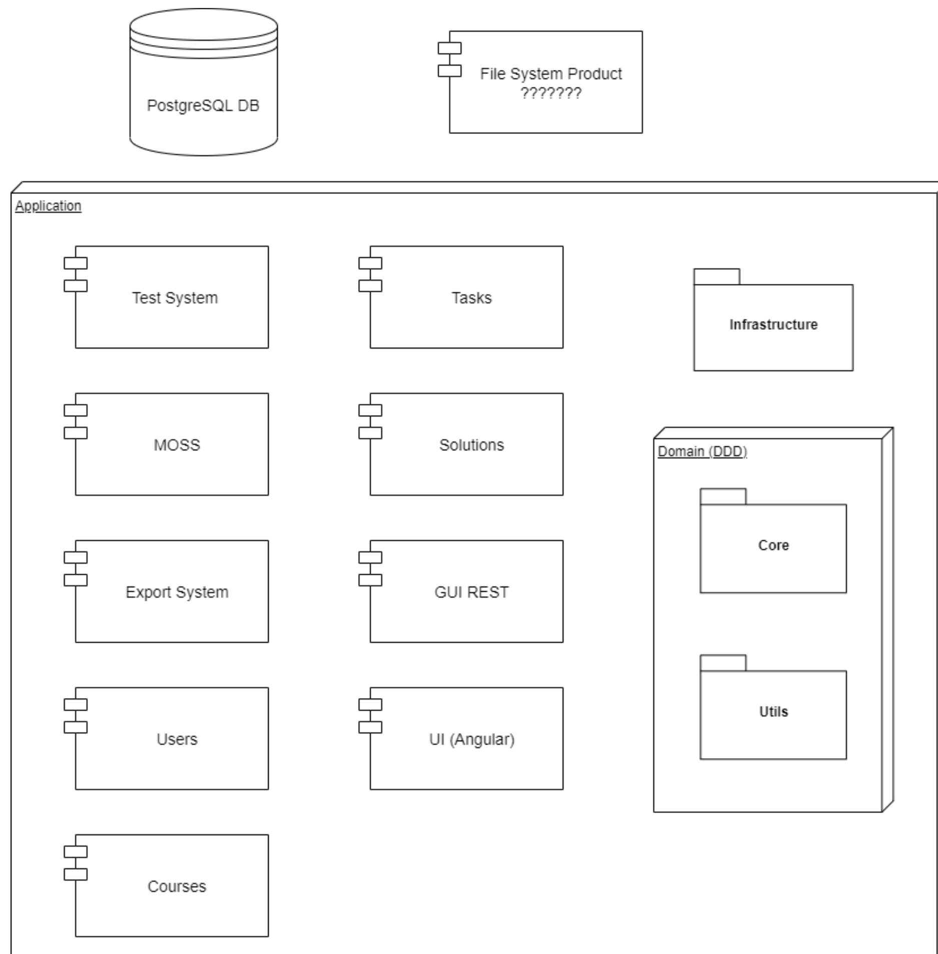
Modularita v prvej iterácii je spracovaná čiastočne chaoticky a najmä nespĺňa skutočnú charakteristiku modularity, nakoľko v module *Access Management System* a *Test System* spravujú viacero nesúvisiacich kontextov v jednej službe.



Obr. 2.1: Prvá iterácia návrhu architektúry a modularity

Druhá iterácia

Prvý návrh bol priamou ukážkou, že rozdelenie domény do skupín, ktoré tvoria moduly, nie je priamočiare. V druhej iterácii návrhu bolo primárnym cieľom zameriať sa na rozdelenie domény na základné moduly. Na obrázku 2.2 vidíme diagram druhej iterácie a najmä prvé pokroky v úspešnom rozdelení domény. Vzniklo tak 6 doménových modulov: Users, Courses, Tasks, Solutions, Test System a MOSS; a zachoval sa API gateway s UI webovou aplikáciou. Značným nedostatkom druhej iterácie je absencia modulov pre pokrytie nutnej funkcionality ako je zaznamenávanie interakcií študentov alebo riešenie pre nahrávanie a načítanie súborov.



Obr. 2.2: Druhá iterácia návrhu architektúry a modularity

Tretia iterácia

Tretia a posledná iterácia zachovala dodatky z druhej iterácie a rieši jeho nedostatky. Pridali sme moduly:

- *Files*: modul manažujúci súbory všetkých typov. Týmto sa rozdelí zodpovednosť modulov tak, aby napríklad modul pre úlohy naozaj spravoval úlohy a potrebné operácie bez potreby ukladania priložených súborov. Taktiež týmto modulom je vyriešená potreba riešiť semantiku a postupnosti ukladania súborov na serveri vo viacerých modulov,

- *Exports*: ako v prvej iterácii, modul má na starosti zozbieranie dát a súborov a vygenerovanie PDF exportu pre úlohy a zostavy úloh,

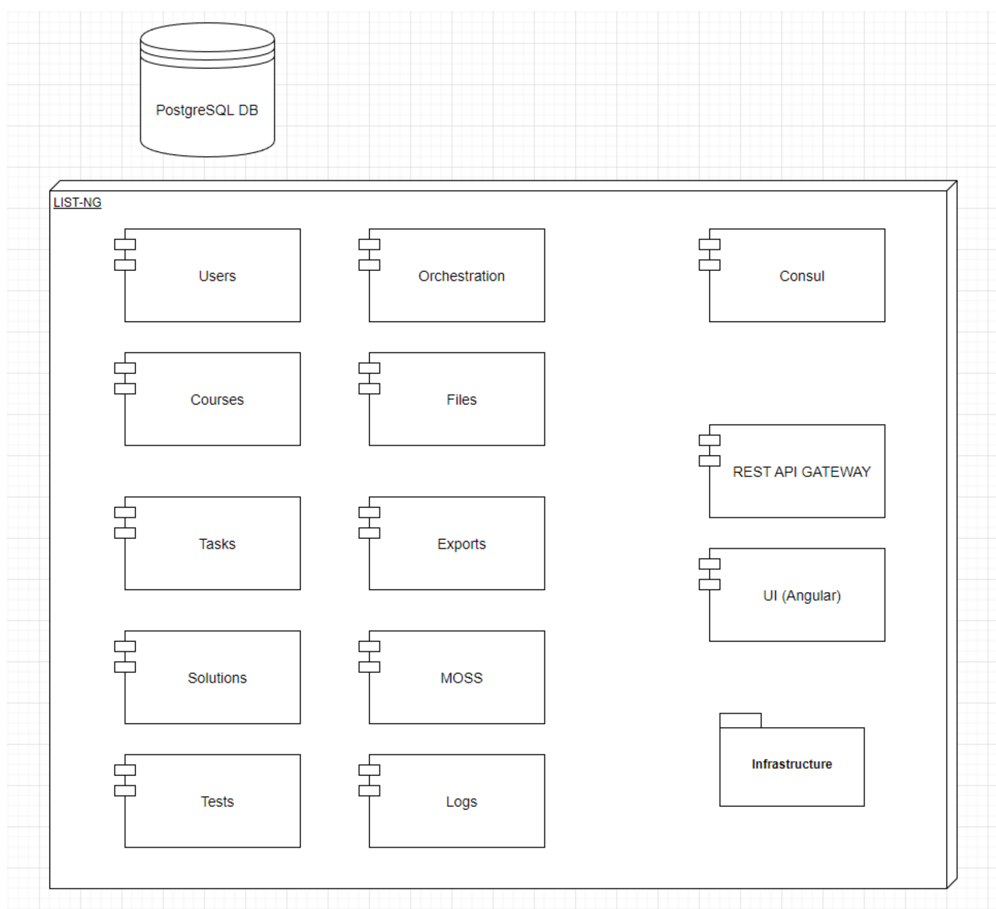
- *Logs*: modul pre zaznamenavanie činnosti v systéme,

- *Orchestration*: modul pre automatizácie a správu systému ako celku. Hlavnou zodpovednosťou modulu je zálohovanie databázy a úložiska súborov, správa inštancii služieb (pridanie/zrušenie inštancii služby),

- *Consul*: modul slúžiaci pre registrovanie a poskytnutie dostúpnych služieb.

Názvy ostatných modulov boli zjednotené s názvami doménových entít, napríklad *Test System* bol premenovaný na *Tests*. Značnou zmenou bolo zrušenie projektov Core a Utils, nakoľko princíp kde by doménové moduly mali závislosť na celé doménové jadro nie je optimálny. Preto každý modul bude obsahovať v sebe aj podprojekt *<MODUL>/domain*, čím sa zbavíme zbytočných tried mimo rozsahu daného modulu (napríklad modul pre správu kurzov nemá potrebu pre triedy a implementácie používateľov).

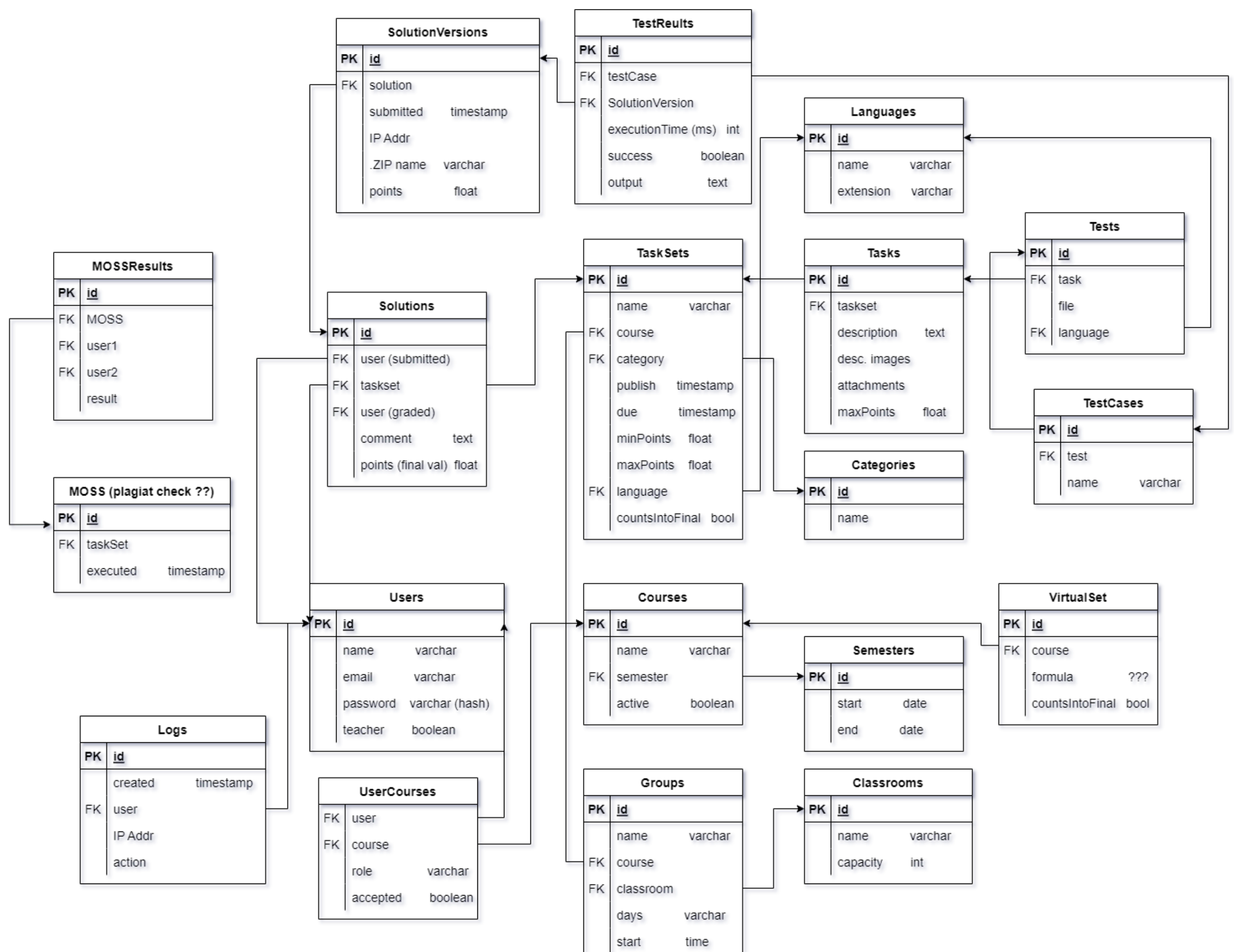
Posledným rozhodnutím a zmenou bolo určenie zodpovednosti knižnice *Infrastructure*. V prechádzajúcich návrhoch projekt slúžil ako knižnica so zdieľanými konfiguráciami služieb a plošné definície technologických závislostí. Tu nám vznikol problém v prípade navýšenia verzie niektorej zo závislosti, ktorá by potencionálne mohla spôsobiť znefunkčnenie všetkých modulov. Preto každý modul bude mať na starosti definíciu vlastných závislostí a projekt *Infrastructure* slúži ako knižnica pre zdieľané implementácie.



Obr. 2.3: Tretia iterácia návrhu architektúry a modularity

2.2.3 Doménový model

Pri návrhu domény sme prešli troma iteráciami ako pri návrhu architektúry. Prvá iterácia návrhu domény, zobrazená na obrázku 2.4, zachytáva základnú funkcionalitu systému LIST odpozorovaná po názornej ukážke a vysvetlení potrebnej logiky. Ako pilotná verzia domény neobsahuje podporu pre ukladanie súborov, všetky reprezentované entity domény sú previazané a nezohľadňuje mnoho požiadaviek. Ako základ domény tvoria entity *User, Course, UserCourse, Task, TaskSet, Test, TestCase, Solution, SolutionVersion* a *TestResult*. Zásadným nedostatkom verzie je vzťah medzi úlohou a zostavou úloh, kde v doméne je úloha vždy priradená práve jednej zostave úloh.

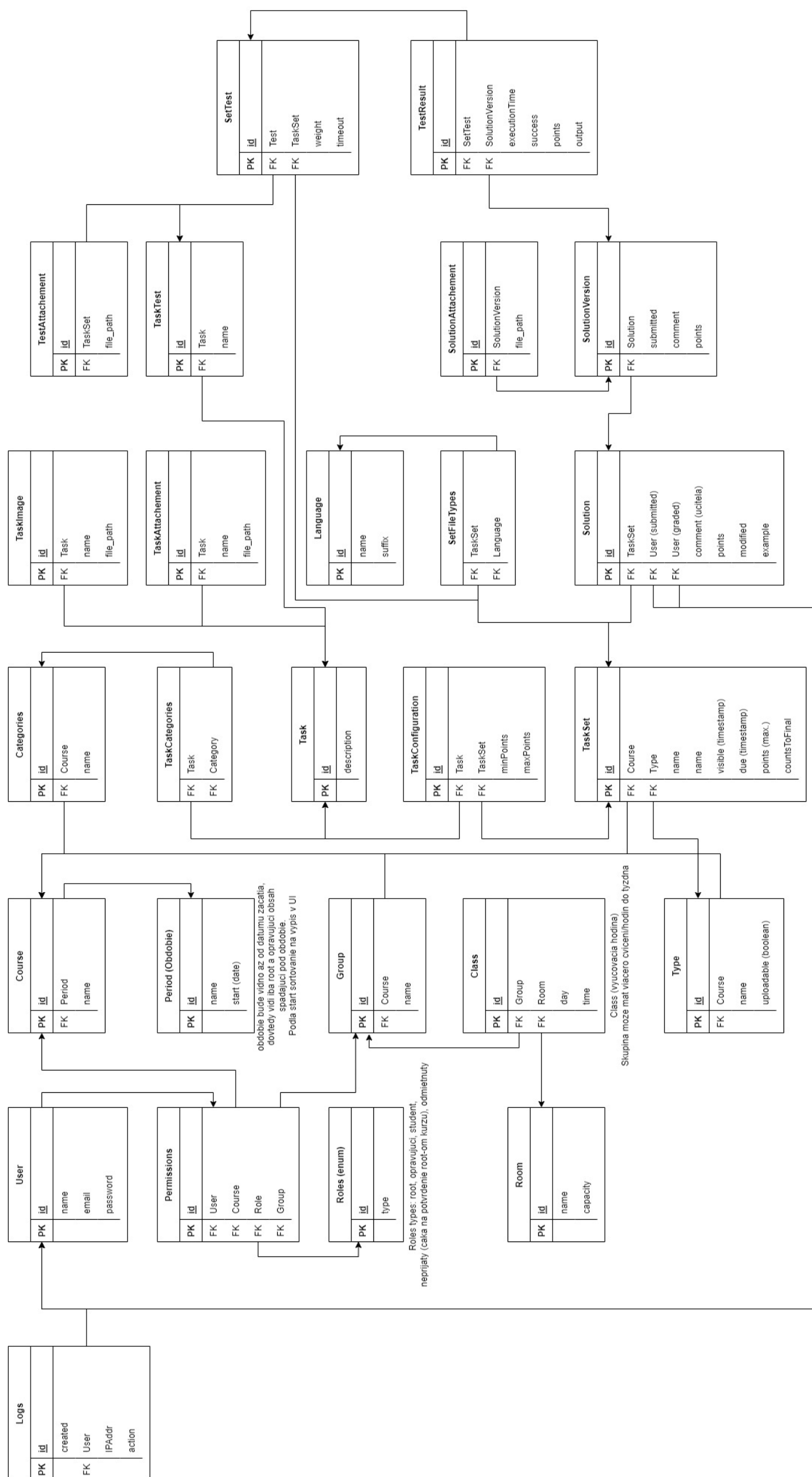


Obr. 2.4: Prvá iterácia návrhu domény.

Bez ohľadu na nedostatky, prvá verzia modelu domény by do systému zaviedla mnoho problémov pri vývoji. Po množstve konzultácií, ďalších stretnutí s ukážkami a štúdii zdrojových kódov LIST-u sme vytvorili druhú iteráciu návrhu domény, ktorý primárne vyriešil absenciu kľúčovej funkcionality. Druhá iterácia na obrázku 2.5 je rozsiahlejšia a väzby medzi entitami sú usporiadanejšie v porovnaní s prvou iteráciou. Hlavnou zmenou je podpora pre ukladanie súborov a ich priradenie k entite, avšak

vidíme množstvo duplicitných tabuliek pre súbory k úlohám, zostavám úloh, testom a verzií odovzdaní. Používateľské práva v tomto návrhu sú odčlenené od entity používateľa, aby sme vytvorili možnosť špecifikovania prístupu ku kurzom jednotlivým používateľom. Naďalej pozostáva nedostatok s úzko spätou doménou, v ktorej bude náročné sa koordinovať a aplikovať potrebné zmeny bez rozpadnutia previazaných entít.

Treťou a poslednou verziou domény sme chceli dosiahnuť zjednodušenie komplexnosti a vytvoriť abstrakciu nad entitami pre rozdelenie domény samotnej podľa modulov. Modulárnou doménou zaručujeme menšiu komplexnosť, jednoduchý prehľad a pochopenie entít a ich vzťahov. Taktiež pri uvažovaní novej funkcionality sa nám ju po analýze ľahko podarí zapojiť. Modularita domény znázornená na obrázku 2.6 je priamou ukážkou takejto výhody. Do tretej iterácie návrhu bola primárne zakomponovaná podpora pre texty s lokalizáciou, jednotná trieda pre predstavenie súborov, obmedzenia pre zostavy úloh a ukladanie používateľského kontextu s nastaveniami v systéme. Tretiu iteráciu návrhu domény bližšie popisujeme v sekcii 2.3.2.



Obr. 2.5: Druhá iterácia návrhu domény.

2.3 Finálny návrh systému

V tejto časti kapitoly bude rozobratý finálny návrh celého systému z hľadiska architektúry, domény, používateľského rozhrania a aj infraštruktúra prostredia pre nasadenie LIST-NG. Finálny návrh predstavuje zamýšľaný cieľový stav, ktorý pokrýva všetky požiadavky ako aj rozšírenia pre LMS systém, ktorý skutočne zlepšuje kvalitu a pohodlie výučby.

2.3.1 Architektúra

Systém LIST-NG sa skladá z 14 projektov: 12 služieb, webová aplikácia a knižnica so zdieľanými implementáciami pre služby. Diagram s presnou skladbou je znázornený na obrázku 2.3, kde projekty a ich zodpovednosti sú:

- *Users*: modul pre správu používateľov, ich oprávnení a uložený kontext. Primárne využitie je overenie prihlásenia pri autentifikácii, správa oprávnení v systéme a v kurzoch a odosielanie email-ových správ. Práve tento modul implementuje odosielanie emailov z bezpečnostného hľadiska, nakoľko pri vytvorení nového účtu sa v pamäti služby generuje náhodne heslo a chceme zabrániť manipulovaniu s jeho hodnotou na iných miestach. Používateľský kontext z aplikácie sa rozumejú posledné otvorené stránky, nastavenia na rôznych miestach a iné, ktoré sa ukladajú do databázy. Pri načítaní aplikácie a relevantnej stránky sa uložené nastavenia kontextu aplikujú.
- *Courses*: modul pre kurzy a vzťahujúce sa aspekty ako obdobia, miestnosti, skupiny a vyučovacie hodiny. Zodpovednosťou je výlučne správa kurzov, období, miestnosti, skupín a vyučovacích hodín.
- *Tasks*: modul pre úlohy a zostavy úloh. Modul spravuje úlohy, kategórie úloh, zostavy úloh, typy zostav úloh, testy pre jednotlivé úlohy a konfigurácie úloh s ich testmi.
- *Solutions*: modul riešení študentov. Modul spravuje jednotlivé riešenia študenta a ich ohodnotenia. Okrem synchrónnej komunikácie prostredníctvom HTTP protokolu, modul je odberateľ topic-u pre asynchrónne správy publikované modulom *Tests*. Pri konzumovaní správ s výsledkami testov tak zaznamená pre danú verziu riešenia jej hodnotenie a aktualizuje najlepšie dosiahnuté skóre študenta v rámci všetkých verzii jeho riešení.
- *Tests*: modul pre automaticky vyhodnocované testy. Modul spravuje asynchrónne vykonáva test pre riešenia zostav úloh a notifikuje modul *Solutions* o výsledkoch. Dôležitou súčasťou je rozumné manažovanie *thread pool-u* pre vykonanie testov

a priradenie priorít testom. Najdôležitejším faktorom je dbať na rôzne prípady, aby všetky dostupné zdroje boli efektívne a spravodlivo alokovane. Modul prijíma požiadavky na vykonanie testu prostredníctvom asynchrónnych správ, spracuje požiadavku a dynamicky alokuje kontajner pre vykonanie testu. V prípade, že všetky požiadavky pre vykonanie testov sú pre práve jeden test, dovolí dočasnej monopole nad kontajnermi. Pri novej požiadavke vykonania odlišného testu, sa jeden z obsluhujúcich kontajnerov pozastaví a modul obslúži novú požiadavku. Takéto dynamické alokovanie je predvoleným správaním, ale ponúka možnosť rezervácie kontajnerov pre zbierku testov v určenom čase (primárne využitie sú skúškové zostavy úloh).

- *Logs*: modul záznamov aktivít. Modul prijíma a spracuje asynchrónne žiadosti pre zápis činnosti, poskytuje ich záznamy a uvoľňuje pamäť v databáze vymazaním starých záznamov.

- *MOSS*: modul kontroly plagiátorstva. Asynchrónne prijíma a spracuje žiadosť pre porovnanie a spravuje výsledky porovnaní.

- *Exports*: modul exportovania. Modul generuje žiadané exporty textov úloh či zostav úloh vo forme PDF spolu s relevantnými prílohami.

- *Files*: modul súborov. Modul spravuje lokálne úložisko súborov všetkých typov, teda prílohy pre úlohy a zostavy, testovacie súbory a súbory z riešení študentov. Pri potrebe súboru v inom module je modul súborov volaný pre vrátenie priamej cesty k žiadaným súborom.

- *Orchestration*: modul administrácie systému. Modul periodicky vykonáva zálohu databázy a úložiska súborov, spravuje stav a konfiguráciu systému.

- *Consul*: register služieb LIST-NG. Automaticky registruje inštancie služieb, spravuje externé property atribúty, poskytuje službám ich priradené atribúty pri registrovaní, vykonáva periodické monitorovanie služieb a poskytuje informácie o zdravých inštanciach služby.

- *REST API Gateway*: API gateway do domény LIST-NG. Modul slúži ako vstupná brána do kontextu LIST-NG, primárne slúži ako jediný verejný koncový bod volaný klientom na požiadavku používateľa s dopytmi pre dáta alebo vykonanie operácií. Modul dopyty spracuje, rozposiela potrebné volania na iné moduly a následne spracuje a spojí odpovede do jednej odpovede pre klienta.

- *UI*: Angular webová aplikácia. Generuje dynamické stránky na žiadosti používateľa spolu s potrebnými skriptmi pre vykonanie logiky na pozadí.

- *Infrastructure*: knižnica pre služby LIST-NG poskytujúca zdieľané implementácie ako jednotný formát odpovedí, spracovanie a pridanie property atribútov a iné.

2.3.2 Model systému

Základ modelu tvorí šesť subdomén, ktoré vidíme na obrázku 2.6. Modely pre moduly *Tests, Exports, Files a Orchestration* niesú potrebné, nakoľko potrebné označujúce dáta sú v iných moduloch. Vysvetlenie domén po moduloch:

- *Users*: Interagujúci používateľ je znázornený triedou *User*, ktorá okrem základných údajov ako meno, email a heslo, obsahuje záznam *SystemRole* priraďujúci oprávnenie nad celým systémom. Študent v systéme sa rozumie používateľ zúčastňujúci sa vyučby ako hodnotený študent alebo ako hodnotiaci a pomocný učiteľ v danom kurze. Učiteľmi sa rozumie používateľ spravujúci chod samotnej výučby vlastných kurzov a okolitých aspektov ako sú obdobia či miestnosti. Rootovský používateľ predstavuje administrátora systému, ktorý je učiteľ s pridaným oprávnením spravovať systém. Oprávnenie ku jednotlivým kurzom nastavujeme pomocou triedy *Permission*, ktorý obsahuje úroveň prístupu ku kurzu (sledovateľ, účastník výučby, hodnotiaci a vlastník) a stav pre oprávnenie samotné. Stav oprávnenia určuje či používateľ má aktívne oprávnenie, očakáva schválenie, žiadosť o oprávnenie bolo zamietnuté alebo bol odstránený z kurzu. Trieda *UserContext* predstavuje uložené nastavenia pre komponenty v používateľskom rozhraní.

- *Courses*: Modul kurzov tvorí malý a jednoduchý model, obsahujúci obdobia, kurzy, skupiny v kurze, miestnosti pre výučbu a opakujúce sa vyučovacie hodiny na týždennej báze.

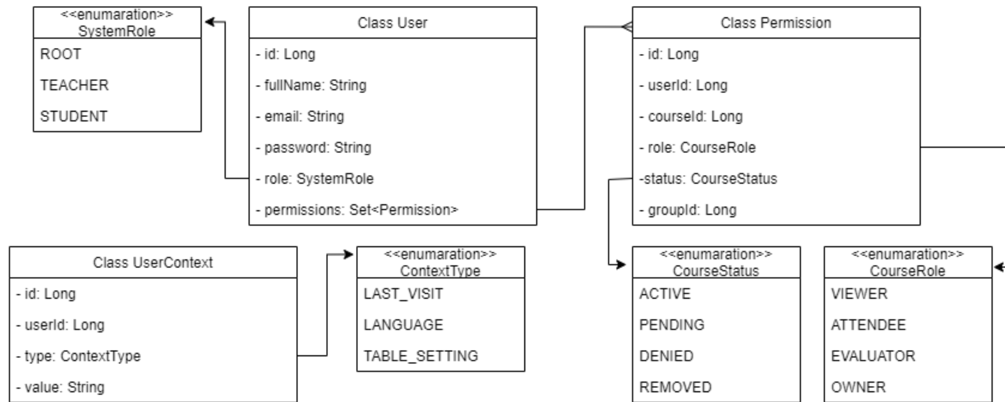
- *Tasks*: Doména úloh obsahuje úlohy samotné spolu so zostavami úloh a ich konfigurácie. Testy samotné sú značnou súčasťou úlohy, preto práve v danom module ich evidujeme a určujeme vzťah ku úlohe. Úlohy sú nepriradené a predstavujú malý celok zadania pre študentov. Pre zaručenie možnosti prepoužitia úloh, v zostave úloh používame triedu *TaskConfiguration* ktorý nám umožňuje totožnú úlohu použiť na viacerých miestach s rozličnými bodmi a verzie testov. Učiteľ pri tvorbe úlohy má možnosť vytvoriť a nahrať mnoho alternatív testov, z ktorých si vyberie ktoré testy budú použité pri automatickom testovaní riešenia. Podobne ako pri konfigurácii úloh vieme vytvoriť rozličné konfigurácie testov v zostave.

- *Solutions*: Modul riešení spracováva jednotlivé riešenia študentov a ich verzie. Trieda *Solution* predstavuje všeobecne riešenie pre zostavu úloh, v rámci kto-

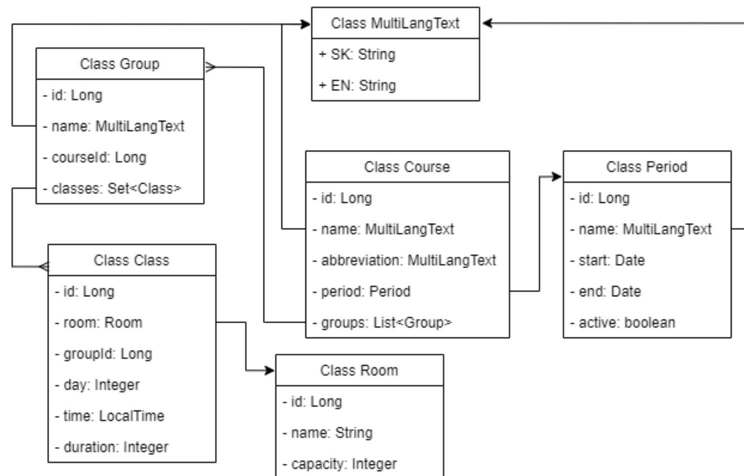
rej vieme evidovať mnoho verzií. Pre jednotlivú verziu riešenia registrujeme aj výsledok automatických testov vykonaných modulom *Tests*.

- *Logs*: Záznamy činnosti používateľov predstavujeme triedou *Log*, ktorá obsahuje informáciu o používateľovi, jeho IP adresy, čas a dátum, typ akcie, statickú správu podľa vykonanej činnosti a detailnú správu popisujúcu činnosť. Typy činnosti rozdeľujeme pomocou enum-u *ActionType*, a statické správy k činnosti popisujú činnosť samotnú bez kontextu daného používateľa. Pre stiahnutie riešenia v uplynulom kurze máme špecifickú správu ako je "Stiahnutie riešenia alebo iných súborov v ukončenom kurze.", podľa čoho vieme následne v používateľskom rozhraní presnejšie filtrovať činnosti a uľahčiť tak prehľadávanie.
- *MOSS*: Porovnávanie riešení študentov je spracované systémom MOSS od Stanfordskej univerzity. Modul MOSS príjme požiadavku o porovnanie riešení v určenej zostave úloh, pre ktorú si zozbiera riešenia prostredníctvom modulu *Files*, odošle požiadavku na MOSS systém a uloží jej výsledok vo forme URL adresy. Jednotlivé porovnania sú zobrazované používateľovi, ktorý si ich vyžiadal.

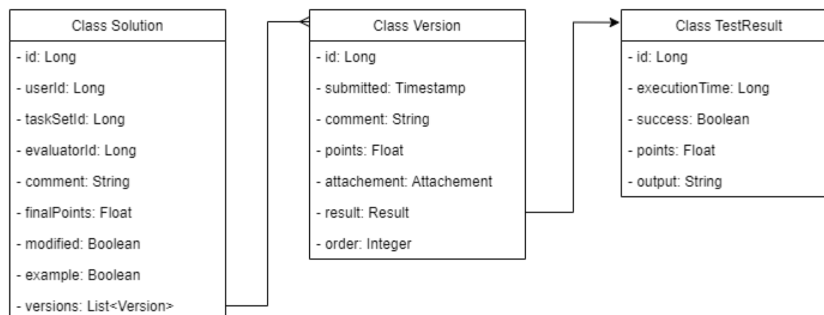
Module Users



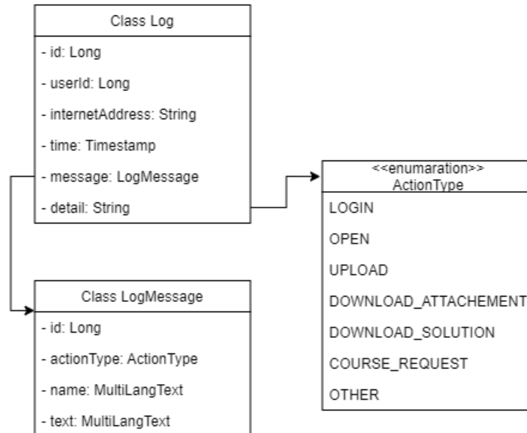
Module Course



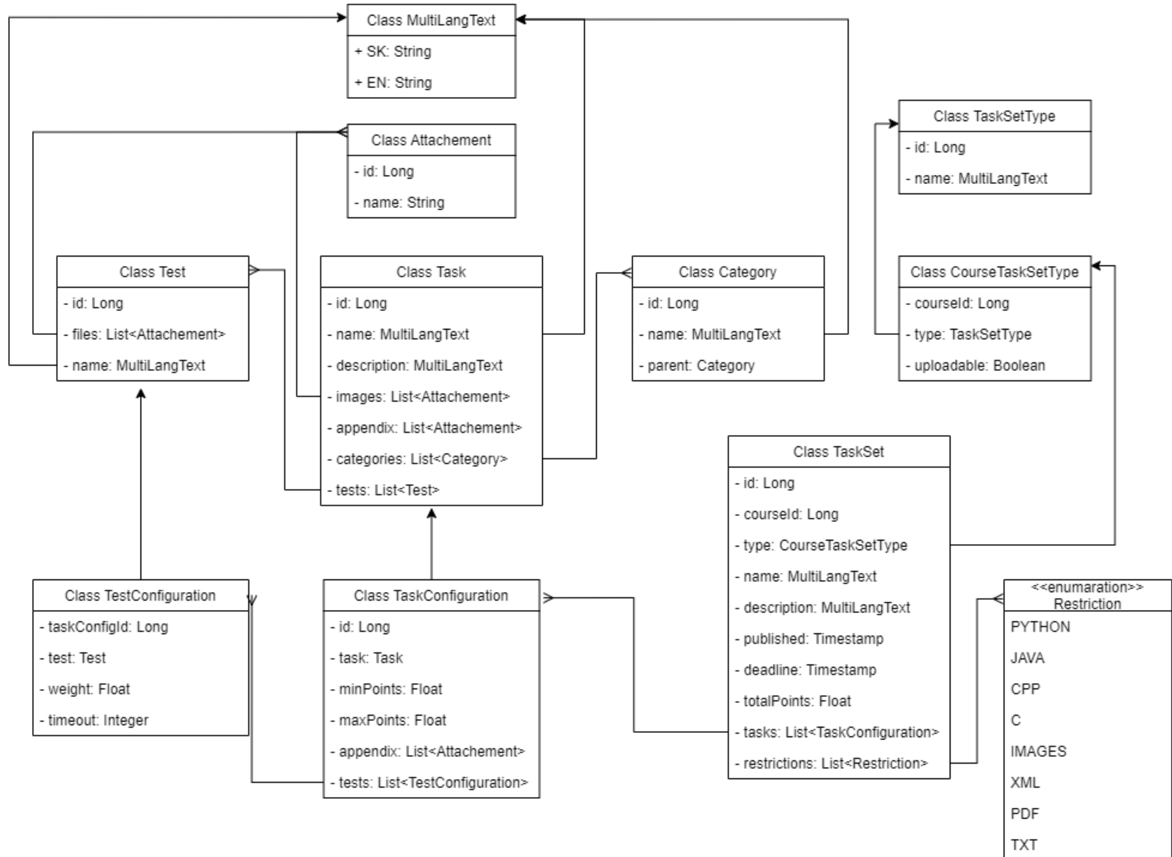
Module Solutions



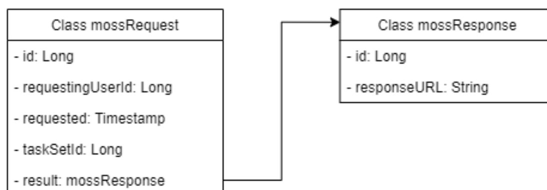
Module Logs



Module Tasks



Module MOSS



Obr. 2.6: Finálny návrh domény.

2.3.3 Používateľské rozhranie

Predchodcovia LIST-NG a ich používateľské rozhranie boli spracované na dve samostatné stránky, jedna slúžila ako rozhranie pre študentov a druhá pre učiteľov a správcov. LIST-NG vďaka jednotnej reprezentácii používateľov v doméne s priradením oprávnení si môže dovoliť rozhrania zjednotiť a tak uľahčiť používateľskú skúsenosť pre vyučujúcich.

Všeobecné rozloženie aplikácie

Na obrázku 2.7 môžeme vidieť všeobecné rozmiestnenie obsahu v aplikácii po prihlásení. Základ vzhľadu tvorí navigačné menu, ktoré pre všetkých používateľov zobrazuje logo v ľavej časti, využité ako tlačidlo pre presmerovanie na domovskú stránku. V pravej časti je tlačidlo na prepínanie medzi jazykmi lokalizácie. Po zmene jazyka lokalizácie sa asynchrónne zmenia texty a použijú sa adekvátne preklady. Položky v prostrednej časti navigácie predstavujú správu záznamov entít. Toto menu sa zobrazuje iba učiteľom a správcom systému, teda k obsahu ani ku funkcionalite študenti nemajú prístup. Overenie prístupu sa vykonáva v rámci API gateway, aby sme zabránili manuálnej úprave atribútov a dát cez konzolu prehliadača. Zoznam dostupných položiek pre vyučujúcich je poskladaný taktiež na API gateway, nakoľko odkaz pre správu celého systému má byť dostupný iba správcom systému a nie všetkým učiteľom.

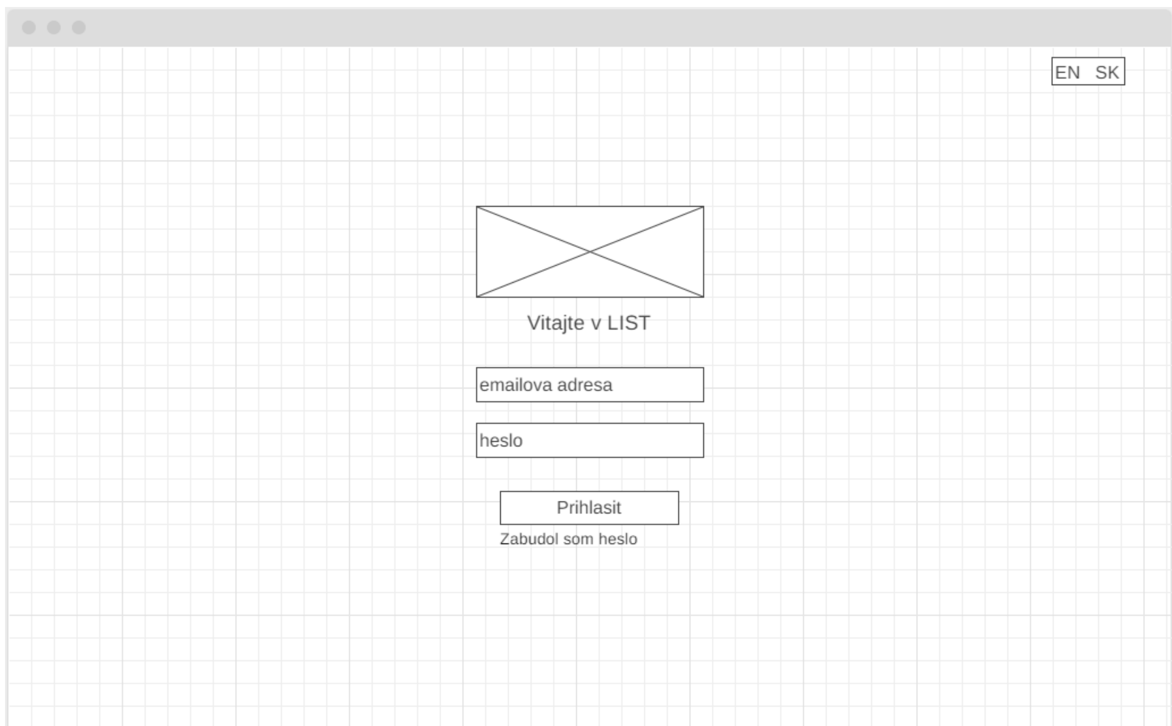
Obdlžník v strede obrazovky predstavuje kontajner v ktorom sa zobrazuje ľubovoľný obsah ale dodržiava jednotný vzhľad aj štruktúru. Kontajner pokrýva 85 percent šírky aplikácie a ponecháva prázdny priestor medzi ním a navigáciou.

Vstup do systému

Vstupnú bránu do webového rozhrania systému tvoria tri okná:

- **Okno pre prihlásenie** zobrazené na obrázku 2.8. Používateľovi sa zobrazuje jednoduchý formulár pre vyplnenie prihlasovacích údajov. Vstup pre jednotlivé údaje majú základnú validáciu, kde obidve sú požadované a vstup pre emailovú adresu musí spĺňať validný formát. V prípade chyby s validáciou alebo pri použití neplatných prihlasovacích údajov sa zobrazí chybová hláška pod vstupmi použitím *mat-err* z knižnice Angular Material [2]. Používateľ má možnosť prepnúť lokalizáciu, prihlásiť sa alebo požiadať o obnovu hesla.

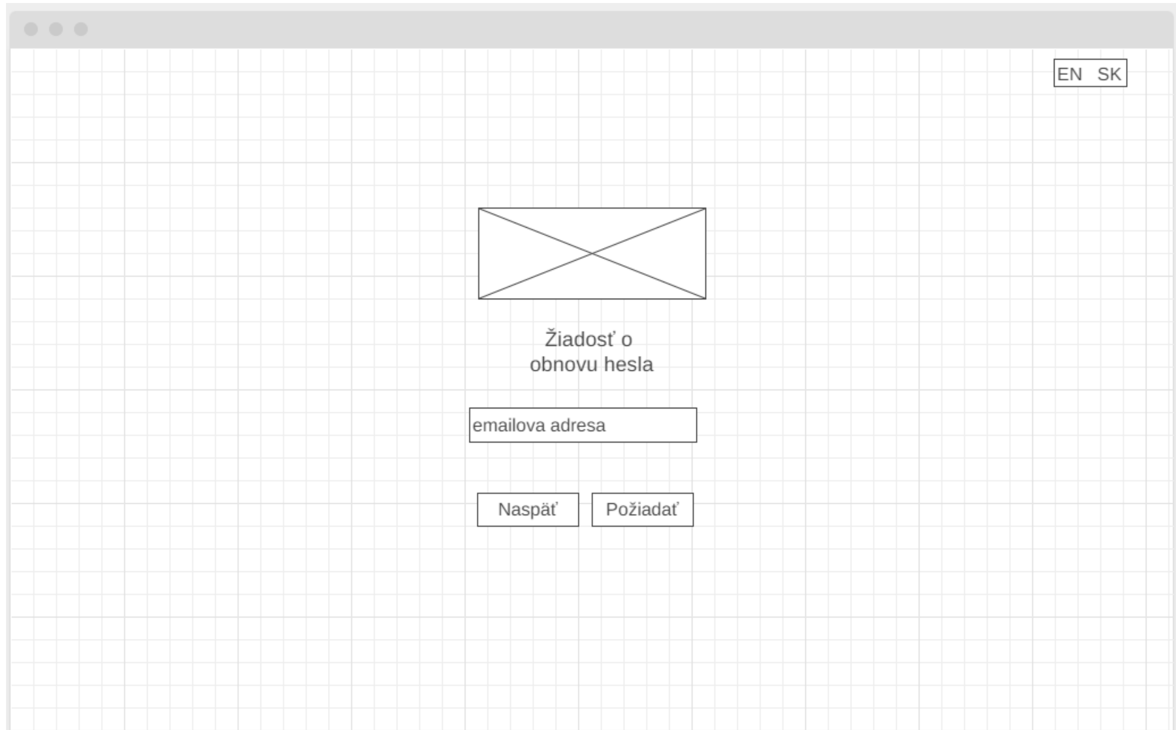
Obr. 2.7: Všeobecné rozloženie obsahu aplikácie



Obr. 2.8: Okno pre prihlásenie

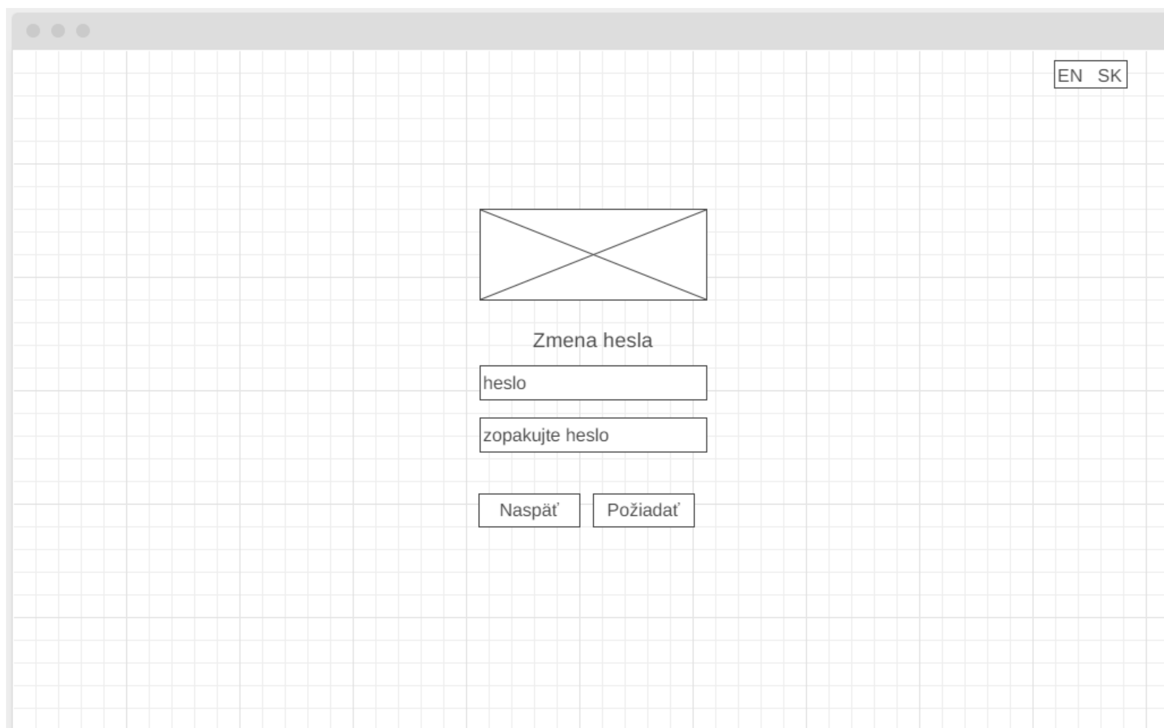
- **Okno pre podanie žiadosti o obnovu hesla** v obrázku 2.9 obsahuje jednoduchý formulár pre zadanie emailovej adresy používateľa. Pri potvrdení žiadosti sa

emailová adresa validuje a v prípade neexistujúceho účtu so zadaným emailom sa používateľovi zobrazí chybová hláška. Po úspešnom potvrdení sa používateľovi odošle email s vygenerovaným odkazom s platnosťou na 15-minút. Generovaný odkaz má platnosť na 15 minút.



Obr. 2.9: Okno pre žiadosť obnovy hesla

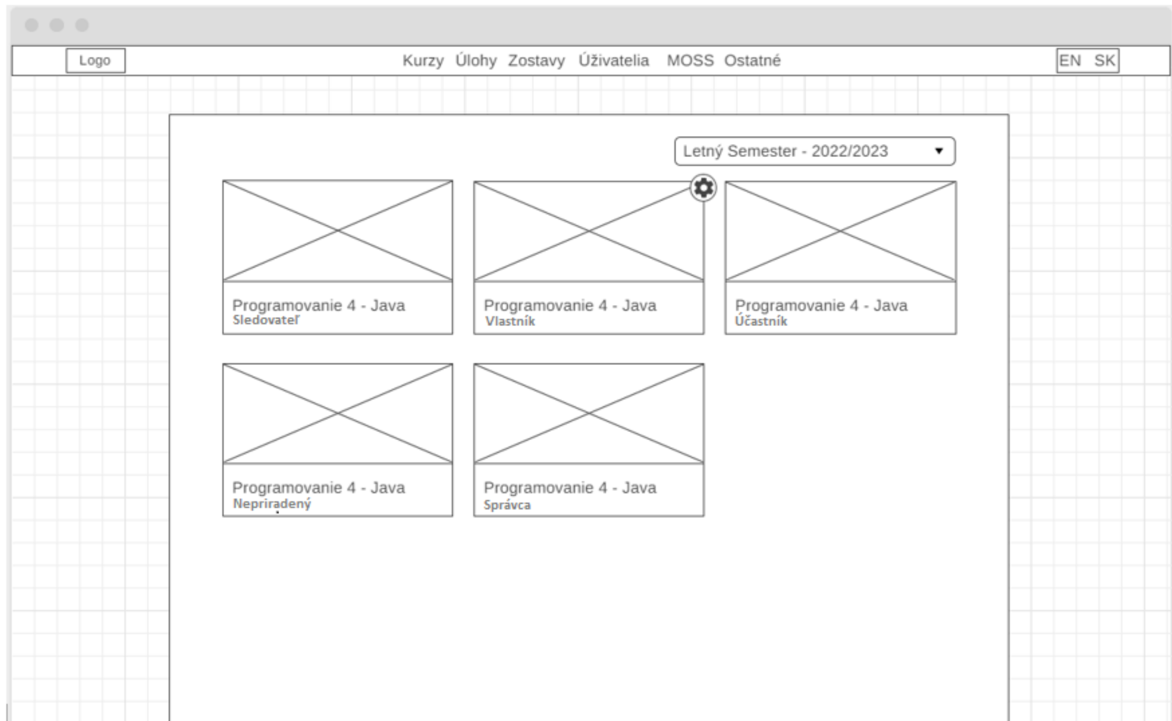
- **Okno pre spracovanie obnovy hesla** v obrázku 2.10 sa skladá z jednoduchého formulára pre zadanie nového hesla. Pri otvorení generovaného odkazu sa validuje jeho správnosť a platnosť, čím zabezpečujeme systém a používateľov voči nekalým praktikám. Formulár pre nové heslo je validované, kde údaje sú požadované, musia byť totožné a heslo musí spĺňať základne podmienky z bezpečnostného hľadiska: aspoň 8 znakov, aspoň jeden špeciálny znak a aspoň jedna číslica. Po úspešnej validácii a spracovaní zmeny je používateľ presmerovaný na okno pre prihlásenie.



Obr. 2.10: Okno pre obnovu hesla

Domovská stránka

Domovská stránka slúži ako rozcestník ku kurzom v zvolenom období. Stránka ponúka menu pre výber obdobia, kde možnosti sú zoradené podľa dátumu začiatku a predvoleným výberom je práve to obdobie, ktoré je označené ako aktívne. Zobrazujeme všetky kurzy v zvolenom období, s popisom oprávnenia používateľa pre špecifický kurz. Oprávnenie v kurze, bližšie popísané v sekcii 2.3.2 vo vysvetlivke pre modul *Users*, vie zmeniť iba vlastník alebo správca kurzu prostredníctvom správy používateľov alebo priamo v prehľade kurzu. Študent pri kliknutí na kurz v ktorom nemá oprávnenie, vie odoslať žiadosť pre pridanie. Po potvrdení žiadosti vlastníkom alebo správcom kurzu sa študentovi umožňuje prístup do kurzu.



Obr. 2.11: Domovské okno aplikácie

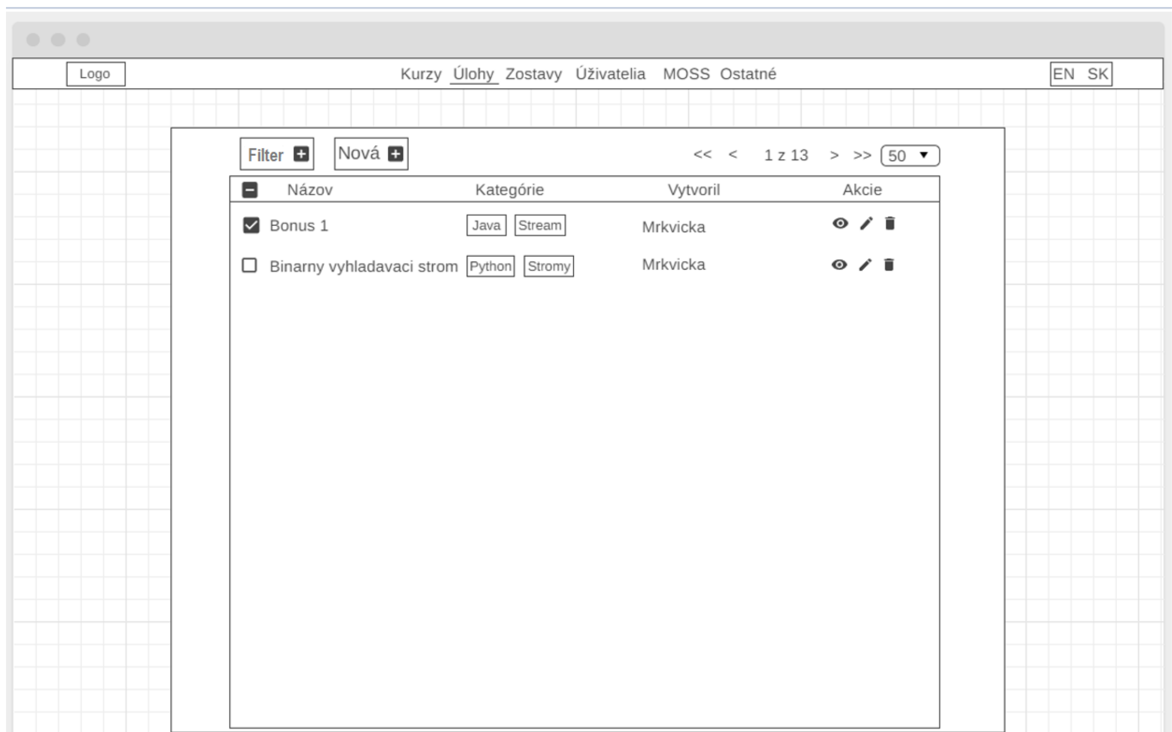
Všeobecné rozloženie okna pre správu

Správa jednotlivých entít je prístupná cez menu v hlavičke stránky, ktoré vidia iba oprávnení používatelia, teda učitelia a správcovia systému. Vzhľad týchto okien pre správu je jednotný pre uľahčenú manipuláciu. V obrázku 2.11 vidíme návrh pre správu úloh, ktorý slúži ako smernica pre vzhľad a skladbu všetkých okien pre správu. Základ stránky tvoria štyri časti:

- Pravá časť sekcie nad tabuľkou obsahuje ovládanie stránkovania a obsahu príslušnej tabuľky. Základné ovládania sú prechody medzi stránkami pomocou ikoniek so šípkami, kde ikonka s jednou šípkou posúva obsah o jednu stránku príslušným smerom a dvojitá šípka posúva obsah na prvú alebo poslednú stránku. Posledným ovládačom je menu s voľbou počtu záznamov pre jednu stránku, kde možnosti sú 50, 100, 200 a 500 podľa žiadosti a špecifikácie správcov LIST-u.
- Ľavá časť sekcie nad tabuľkou predstavuje možnosti interakcie s obsahom tabuľky. Hlavné interakcie ponúkane v tejto časti sú filtrovanie obsahu, hromadné akcie nad výberom ako je úprava alebo vymazanie, vytvorenie novej entity a iné podľa potreby kontextu.
- Hlavička tabuľky obsahuje popis stĺpcov, ktoré vieme kliknutím aplikovať zoradenie podľa príslušného stĺpca. Zoradiť vieme oboma smermi, teda zostupne a vzostupne. Pre entity s možnosťou hromadnej interakcie máme začiarkavacie

políčko pre označenie alebo odznačenie všetkých entít zobrazené na aktuálnej stránke tabuľky.

- Telo tabuľky tvorí prefiltrovaný a zoradený obsah. Všeobecná skladba riadkov je začiarkavacie políčko, stĺpce s dátami a možné operácie nad danou entitou v riadku. Základné akcie sú úprava a odstránenie. Ďalšie operácie sa líšia podľa kontextu, napríklad v prípade úloh a zostáv je možnosť otvoriť náhľad aký vidia aj študenti medzi zadaniami.

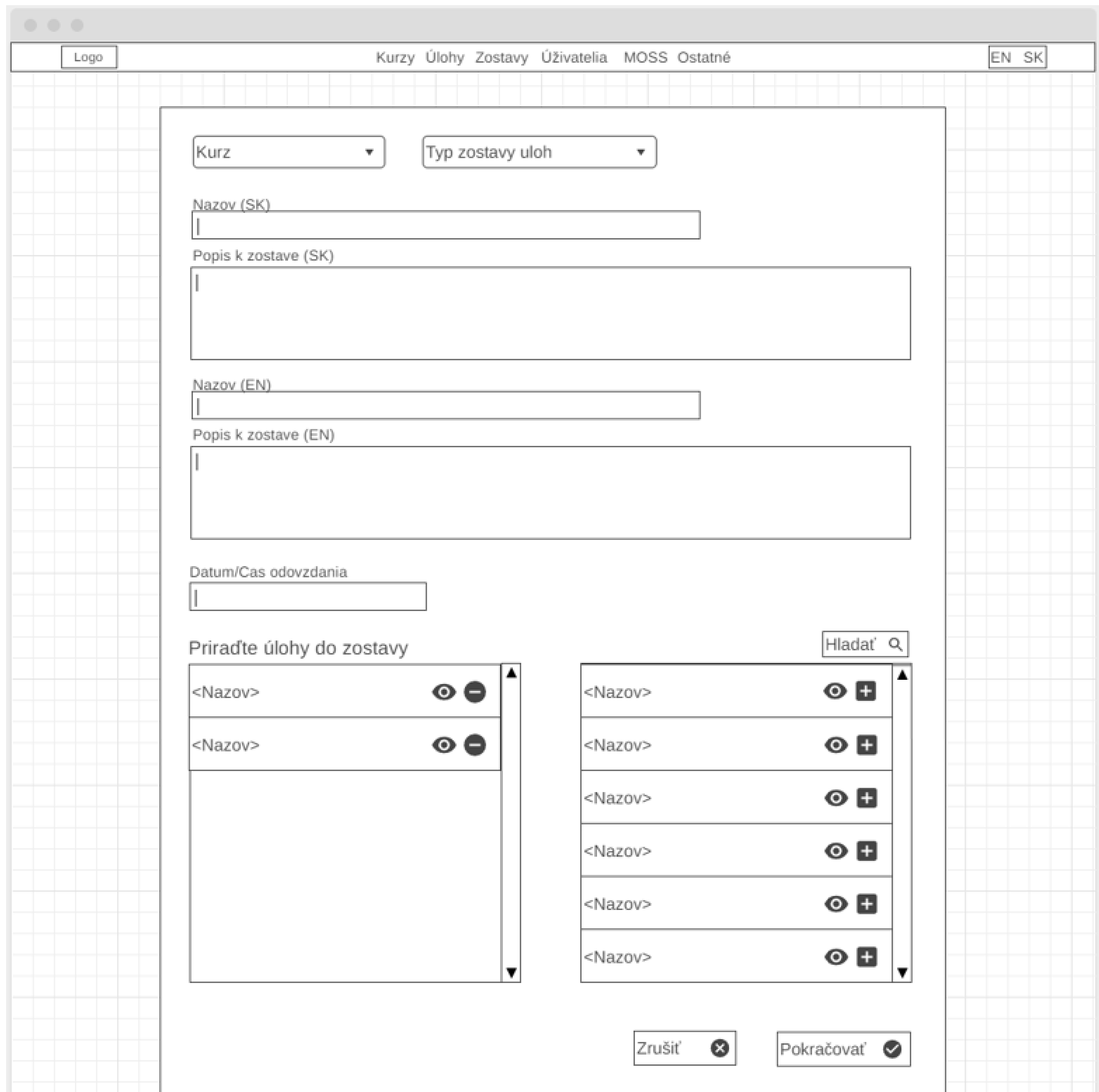


Obr. 2.12: Všeobecné rozloženie okna so správou

Vytvorenie a úprava zostavy

Zostavy úloh sú najdôležitejším aspektom systému, nakoľko predstavujú zadanie na vypracovanie študentmi. Obrázok 2.13 zobrazuje okno s formulárom pre vytvorenie novej zostavy úloh. Pre vytvorenie zostavy úloh je povinné vybrať kurz a typ zostavy úloh, zadať názov a pridať úlohy. Výber úloh je implementovaný prostredníctvom zoznamu úloh, ktorý vieme filtrovať po kliknutí na tlačidlo "Hľadať". Po kliknutí sa zobrazí modálne okno s možnými filtrami, ktoré sa aplikujú po ich potvrdení. Pridať úlohu do zostavy vieme potiahnutím do zoznamu predstavujúceho výber úloh v zostave, alebo stlačením tlačidla s ikonkou "plus". Po potvrdení základných údajov a výbere úloh sa zobrazí ďalšie okno kde je potrebná konfigurácia úloh. Úlohy môžu mať priložené viaceré súbory pre študentov a testy. V časti konfigurácie úloh sa označujú publiko-

vané súbory ako prílohy, použité testovacie súbory pri vykonaní testov, minimálne a maximálne dosiahnuteľné body.



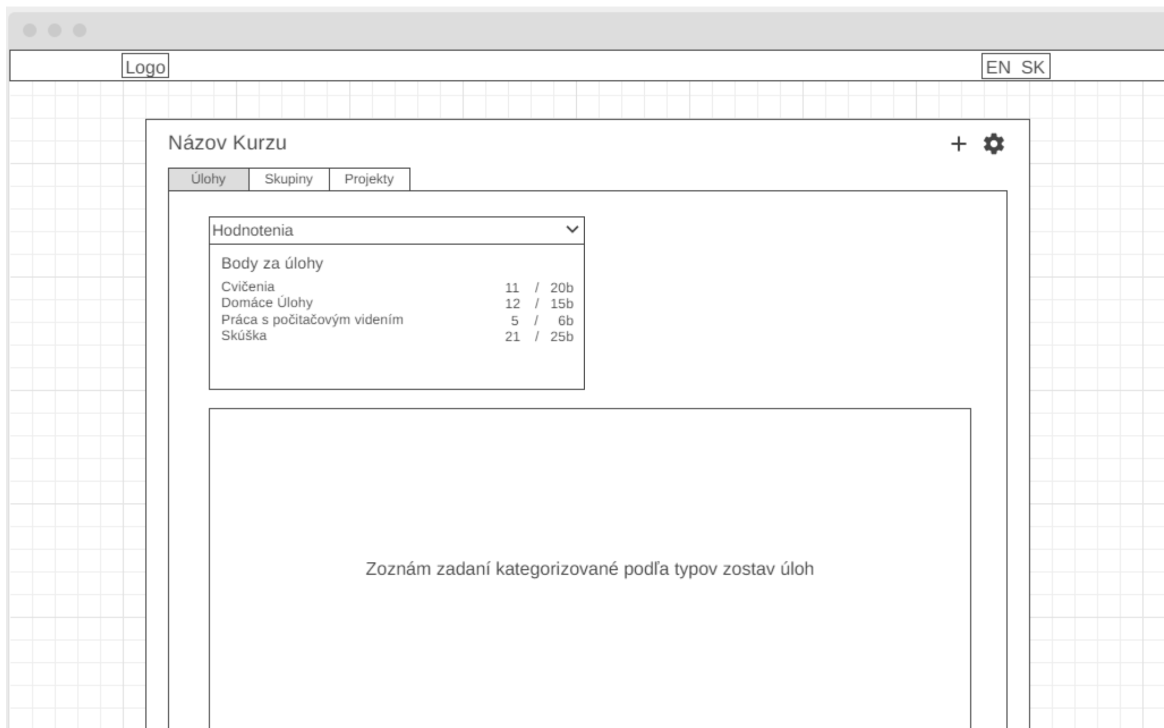
Obr. 2.13: Vytvorenie zostavy úloh

Prehľad kurzu

Hlavné rozhranie spočíva v domovskej stránke a okne s prehľadom kurzu. Návrh okna prehľadu v obrázku 2.14, ponúka študentom primárnu úlohu systému ako je prehľad hodnotení a odovzdávanie riešení pre zadania. Pre používateľa, ktorý je správcom daného kurzu, prehľad navyše obsahuje tlačidlá na príslušné okná pre správu a riadenie kurzu, ako napríklad úprava názvov, skupín a vyučovacích hodín alebo vytvorenie novej zostavy úloh pre otvorený kurz. Okno samotné je členené do troch kontextov:

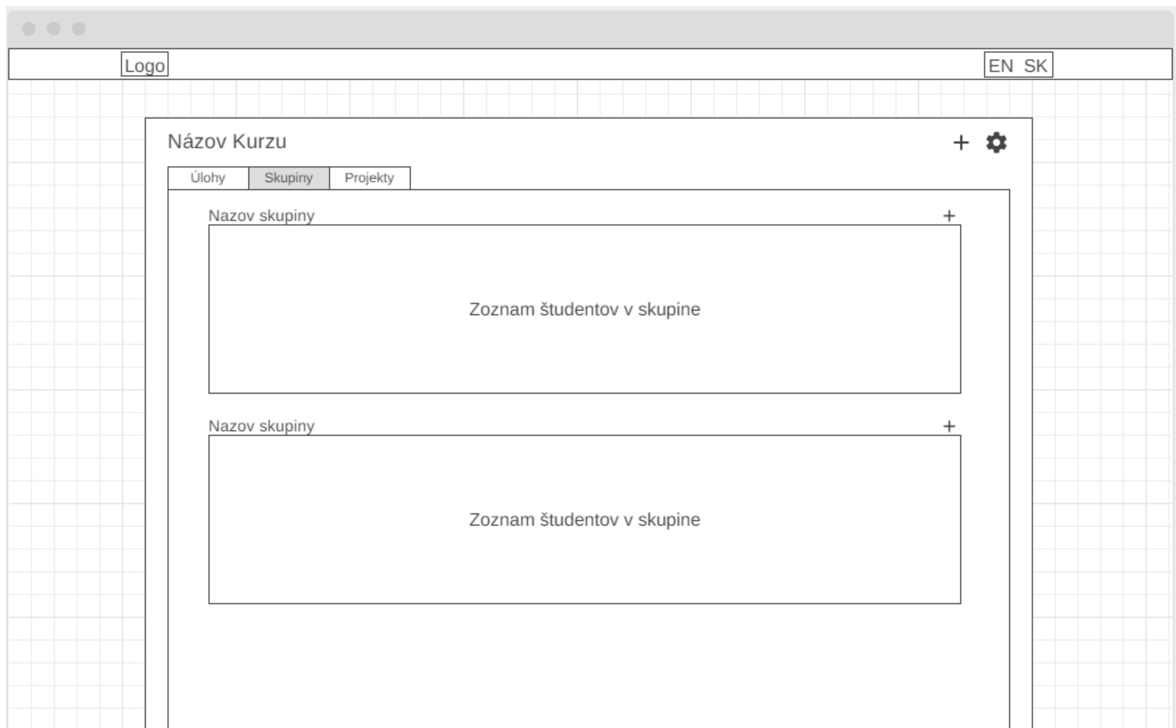
- **Kontext úloh a hodnotení** na obrázku 2.14 obsahuje prehľad celkových hod-

notení a zoznam prístupných zostav úloh. Sekcia s hodnoteniami je skrytá a dá sa rozbaliť kliknutím na hlavičku sekcie. Hodnotenia sú zoskupené a sčítané podľa typu zostav úloh a taktiež obsahuje sekciu s kritériami pre úspešné absolvovanie kurzu.



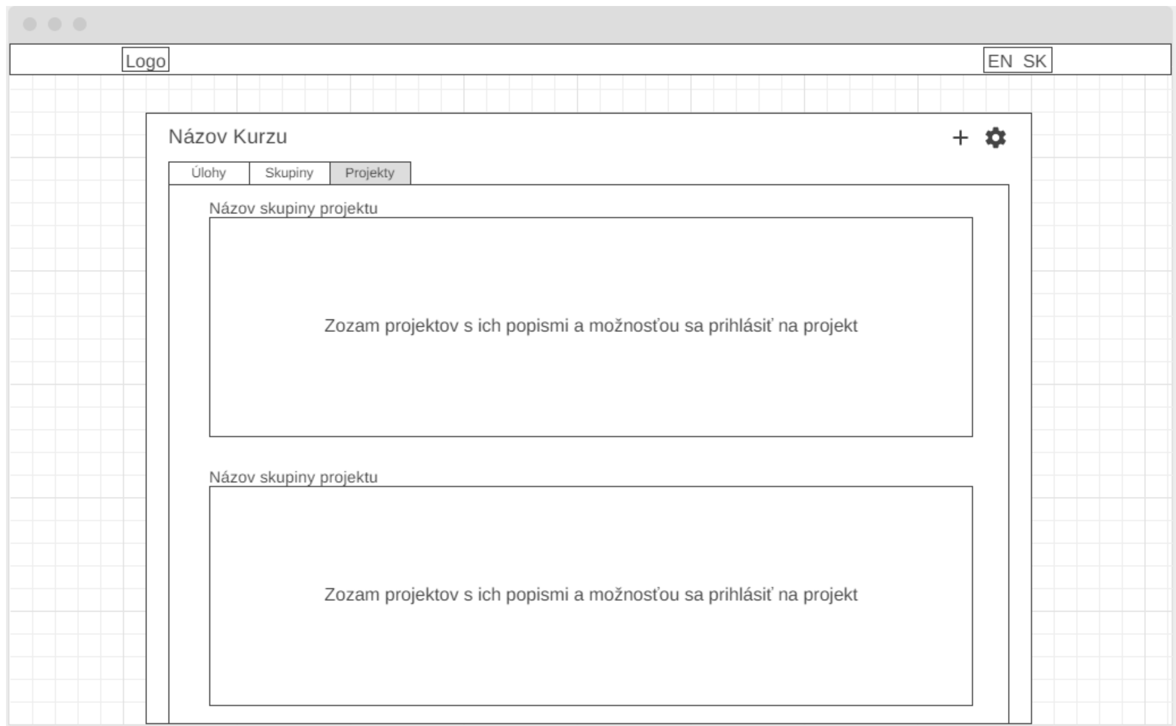
Obr. 2.14: Prehľad úloh a hodnotení v kurze z pohľadu správcu kurzu.

- **Kontext skupín** na obrázku 2.15 zobrazuje rozdelenie študentov do skupín v kurze vo forme zoznamu mien. Študent si vie zmeniť skupinu pokiaľ je úprava skupín ešte povolená alebo správca kurzu vie označiť mená študentov a premiestniť ich do inej skupiny.



Obr. 2.15: Prehľad skupín v kurze s možnosťou zmeny vlastnej skupiny.

- **Kontext projektov** na obrazovke 2.16 zobrazuje témy pre projekty, z ktorých si študent vyberá špecifické zadanie na vypracovanie. Kurz môže vyžadovať študentov, aby vypracovali viacero projektov v priebehu výučby kvôli čomu sme zaviedli členenie projektov do skupín. Každá téma má počet voľných miest a študenti, ktorí si ju vyberú obsadzujú voľné miesta. V každej skupine si vie študent vybrať práve jednu tému. V zozname projektov študent vidí názvy a môže si projekt otvoriť pre náhľad zadania s popisom pre vypracovanie. Po výbere zadania pre projekt sa automatický pridá téma projektu do tabuľky zostav pod typom "Projekty". Obsah vytvorenej úlohy je popis zadania spolu s potrebnými materiálmi pripojené ku zadaniu. Úloha slúži aj ako miesto pre odovzdanie riešenia.



Obr. 2.16: Prehľad projektov, dostupných zadaní a prihlasovanie.

2.3.4 Infraštruktúra

Prostredie pre nasadenie systému je Ubuntu server s nainštalovaným LTS verziami Java 17, NodeJs 18.16.0 a Postgres 15. Databáza beží na predvolenom porte 5432 ale prístup je obmedzený na dopyty z localhost-u. Server je zabezpečený s firewall-om, kde všetky porty zo systému sú blokované okrem portov pre SSH prístup (22), HTTP (80), HTTPS (443), Angular aplikácia (4200) a REST API gateway (8080-8089). Pre jednotlivé moduly sú vyhradené rozpätia portov pre jednoduchú správu:

- *Users*: 8040 - 8049
- *Courses*: 8050 - 8059
- *Tasks*: 8060 - 8069
- *Solutions*: 8070 - 8079
- *Tests*: 8090 - 8099
- *Logs*: 9000 - 9009
- *MOSS*: 9010 - 9011
- *Exports*: 9012 - 9013

- *Files*: 9015 - 9019
- *Orchestration*: 9014
- *Consul*: 8500
- *REST API Gateway*: 8080 - 8089
- *UI webová aplikácia*: 4200

Pre moduly ako orchestrácie, Consul a Angular aplikácia postačuje jediná inštancia teda nie je potrebné vyhradiť viac než jeden port. Určené rozpätie má zaznamenaný modul pre orchestráciu, aby vedel pri dopyte o vytvorenie ďalšej inštancie ľubovoľného modulu určiť jeho port pri nasadení. Nakoľko používateľ priamo komunikuje iba s REST API gateway a UI aplikáciou, vieme všetky ostatné porty zablokovať pre vonkajšie dopyty, čím zaručíme bezpečnosť systému.

Systém používa jednu databázu pre pokrytie celého modelu, avšak každý modul má vlastný účet s prístupom na jedinú vlastnú schému. Rozdelením databázy na schémy zaručíme modularitu a zabránime možnému problému s integritou dát keď viacero modulov k ním pristupuje súčasne.

Moduly samotné sú nasadené a spustené ako bežné služby bez potreby ich registrácie na webový server ako je Tomcat. Pre prístup do aplikácie klient komunikuje prostredníctvom štandardného HTTPS portu 443, na ktorom beží webový server Nginx s nakonfigurovaným reverse proxy. Reverse proxy presmeruje dopyty klienta podľa adresy na príslušný modul, ktorý určujeme podľa koreňovej cesty dopytu. Pokiaľ dopyt začína s cestou "*<DNS>/api*"presmerujeme dopyt na inštanciu REST API gateway, ostatné sú presmerované na UI aplikáciu.

Kapitola 3

Realizácia a implementácia

V tejto kapitole sú zdokumentované implementácie systému s popisom jednotlivých krokov a odôvodnení.

3.1 Doména a služby

Pre implementáciu služieb systému LIST-NG sme zvolili ako základ Spring framework, ktorý je implementovaný v jazyku Java. Pre prácu s objektami a ich mapovanie do relačnej databázy využívame nástroj Hibernate ORM. Ako základ komunikácie medzi službami využívame HTTP protokol so štandardom REST, ktorým spracujeme synchrónne a asynchrónne dopyty.

3.1.1 Štruktúra projektu služby

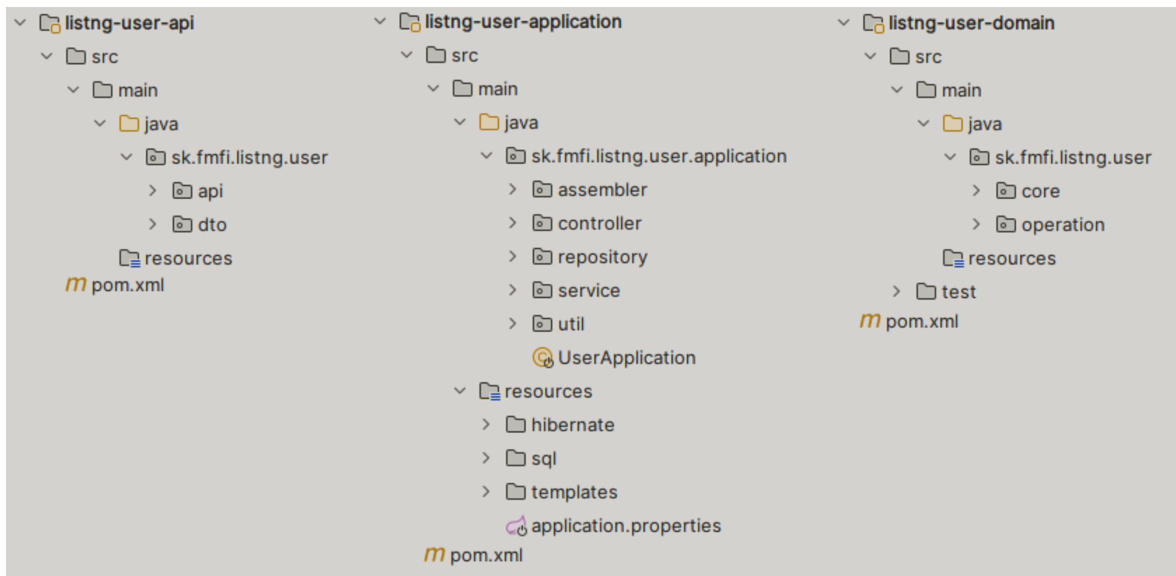
Projekt jednotlivých služieb so zodpovednosťou správy časti domény sme rozdelili na podmoduly pre zakomponovanie hexagonálnej architektúry s princípom Domain Driven Design (DDD). Príklad projektu služby na obrázku 3.1, na ktorom je zobrazený projekt pre modul používateľov, ukazuje skladbu a rozdelenie modulu na tri časti:

- *Domain*: modul s balíkmi implementáciami entitných tried a doménové operácie nad nimi. Modul slúži ako jadro v hexagonálnej architektúre, ktoré obsahuje všetky potrebné implementácie a operácie nad doménou. Integrita domény je zaručená pomocou unit testov nad jednotlivými implementáciami. Modul domény obsahuje výlučne technologické závislosti potrebné pre dosiahnutie doménovej logiky nad entitami.

- *API*: verejný balík, ktorý obsahuje rozhranie služby a jej dostupné mapovania pre dopyty. Okrem rozhraní obsahuje aj reprezentačné triedy pre entity v doméne bez žiadnych implementácií. Týmto sa využil návrhový vzor *Data Transfer Object (DTO)*, kde reprezentačné triedy slúžia ako objekt pre prenášanie dát medzi

procesmi bez prístupu k implementáciám. Tento modul je využitý ako závislosť v inej službe pri potrebe nadviazania komunikácie.

- *Application*: modul implementujúci API rozhranie. Obsahuje controller triedy, ktoré sú priamou implementáciou API rozhraní, ktorých úlohou je prijať dopyt, volať príslušnú servisnú triedu a odoslať jej transformovanú odpoveď. Pre operácie s databázou má modul v repository rozhrania pre jednotlivé entity ktoré rozširujú Spring rozhranie *JpaRepository* [32]. Rozhranie zo Spring framework nám pre danú entitu umožni CRUD operácie a vytvorenie vlastných dopytov metódami s určenou konvenciou pomenovania [6]. Ukážkou repository rozhrania s vlastnými metódami vidíme v ukážke kódu 3.1, kde pri správnom pomenovaní metódy vieme získať požadovanú implementáciu. Ďalej podmodul obsahuje servisné triedy, assembler triedy pre transformáciu objektu do DTO, hibernate mapovania entít v XML súboroch a v prípade potreby SQL skripty pre vytvorenie domény modulu.



Obr. 3.1: Štruktúra projektu služby.

```
public interface UserRepository extends JpaRepository<User, Long>{

        User findByEmail(String email);

        List<User> findAllByIdIn(List<Long> userIds);

        boolean existsByEmail(String email);

        void deleteById(Long userId);

        void deleteByIdIn(List<Long> userIds);

        Page<User> findAllByRoleIn(Pageable pageable, List<SystemRole>
    roles);
    }
```

Kód 3.1: Vzor repository rozhrania pre vykonanie CRUD operácii nad databázou.

Služby obsahujú jednoduchú spustiteľnú triedu, ako príklad v kóde 3.2 používame hlavnú triedu pre modul používateľov. Trieda je označená Spring anotáciami pre jej označenie ako hlavnú konfiguráciu a povolenie implementácie registrovania do služby Consul. Okrem anotácií v main metóde voláme statickú metódu z infrastucture, ktorá pre službu nastaví všetky potrebné atribúty konfigurácie pre správnu registráciu v Consul a označenie mapovacích XML súborov pre Hibernate.

```
@SpringBootApplication
@EnableDiscoveryClient
public class UserApplication {

    public static void main(String[] args) {
        PropertyInitializer.beforeSpringApplicationRun();
        SpringApplication.run(UserApplication.class, args);
    }
}
```

Kód 3.2: Spustiteľná trieda služby.

3.1.2 Implementácia zdieľanej knižnice

V moduloch systému sme zaviedli jednotnú reprezentáciu dát pomocou princípu DTO [5] a potrebu pre totožnú funkcionalitu na rôznych miestach. Preto sme vytvorili knižnicu *Infrastructure*, v ktorej sme implementovali funkcionalitu potrebnú vo viacerých moduloch, aby sme sa vyhli duplicite kódu.

Služby v systéme majú určité a totožné nutné konfiguračné atribúty, ktoré by sa mali nachádzať v *application.properties* danej služby. Kvôli problému s duplicitou konfigurácií sme implementovali triedu *PropertyInitializer*, ktorej úlohou je nastaviť službe hodnoty atribútov. V triede sme vytvorili metódu pre inicializáciu Consul atribútov a ďalšiu pre vyhľadanie Hibernate mapovacích súborov, ktorých názvy sa zreťazia a nastavia ako hodnota pre atribút *spring.jpa.mapping-resouces*. Samozrejme je nutnou podmienkou, že projekt modulu spĺňa jednotnú štruktúru a jej mapovacie súbory sa nachádzajú v adresári *'resources/hibernate'*.

Pre doménové entity s atribútmi názvu alebo textu, ktorým chceme umožniť preklad podľa lokalizácie, sme vytvorili doménovú a DTO triedu *MultiLangText* obsahujúci atribúty *SK* a *EN* pre reprezentáciu slovenského a anglického prekladu reťazca.

Pre aplikačnú časť služieb sme implementovali jednotný formát reprezentácie odpovede stránky pre tabuľku so stránkovaním s triedou *PageResponse* a potrebné parametre pre dopyt o stránku *PagingParams* a *SortParams*. Odpoveď so stránkou obsahuje zoznam dát, číslo stránky, počet všetkých dát dostupné pre dopyt s parametrami a celkový počet stránok. Parametre samotné pre dopyt o stránku obsahuje zoznam kritérií pre zoradenie (stĺpec a poradie), číslo žiadanej stránky a počet záznamov v stránke. Pre stránkovanie chýba zdieľaná reprezentácia filtrov, ktorú sme sa neúspešne pokúsili implementovať pomocou generických tried a predikátov. V rámci rozhrania *JpaRepository* je možnosť implementácie takýchto predikátov pre aplikovanie filtrovania na úrovni databázového dopytu [30]. V systéme LIST-NG chceme podporu filtrovania pre rovnosť, nerovnosť, inklúziu, menší než a väčší než nad atribútmi, podľa ktorého sme vytvorili enumeračnú triedu *FilterOperation* s týmito možnosťami ako aj *FilterAssembler*, ktorého úlohou je vytvorenie generického Java predikátu podstrčého repository rozhraniu, ktorý ho vie preložiť ako podmienku pre SELECT dopyt. Žiaľ všeobecné spracovanie filtrovania sa nepodarilo kvôli potrebe upresnenia typu entity priamo v predikátoch.

Najdôležitejšou implementáciou v knižnici je zavedená jednotná reprezentácia odpovede pomocou triedy *Response*. Trieda okrem dát v atribúte *payload* obsahuje atribúty pre reprezentáciu počtu dát, úspešnosti dopytu a chybovej hlášky. Triedu využívame ako obálku pre odpovede všetkých dopytov v systéme, vďaka čomu vieme vo volajúcej službe spracovať chybu podľa potreby.

3.1.3 Consul

Využili sme Consul pre správu ďalších modulov, ktorý sme zabalili do Spring-ovej aplikácie pre možnosť spustenia na localhost-e pri vývoji. Modul obsahuje rozširujúce implementácie ako naštartovanie aplikácie zabalené v Spring aplikácii, určenie master node Consul-u, inicializácia konfiguračných atribútov pre jednotlivé moduly, nastavenie prístupu a registráciu služieb. Taktiež sme implementovali možnosť registrácie služieb

nasadenú na serveri pre odľahčenie vývojára, aby nemusel mať lokálne spustené všetky moduly pri vývoji.

3.1.4 Controller a mapovania

Pre vytvorenie dopytu na službu a jej špecifickú metódu sme implementovali takzvané controller triedy. Okrem modulu REST API gateway tieto triedy implementujú API rozhrania, ktoré obsahujú definíciu dopytov. Ako príklad API vidíme v kóde 3.3 ukážku API a jej implementácie. V API rozhraní sme definovali jednotlivým metódam ich mapovania, teda typ HTTP metódy a kontextovú cestu čo určuje skladbu URL adresy pre ich využitie. Výsledná URL cesta je poskladaná z IP adresy s portom, kontextová cesta služby, controller-a a metódy, čo určuje službu, jej API a metódu ktorú chceme volať. Kontextová cesta pre API a controller je definovaná priamo controller triede, ktorá okrem implementovania rozhrania je označená anotáciami *@RestController* a *@RequestMapping("<PATH>")* s hodnotou cesty jej kontextu. Napríklad pre implementáciu metódy PeriodApi\#create výsledná URL je <IP>:<PORT>/course/period/new.

Postupne sme analyzovali služby kurzov, používateľov, úloh a vytvorili im API rozhrania s definíciami mapovaní, využili triedu Response ako obálku pre všetky odpovede implementovali metódy. Pre kvalitu kódu a dodržanie hexagonálnej architektúry sme implementovali mapovania sémantikou, kde controller prijíma dopyt, volá príslušnú servisnú triedu, ktorá získa z repository rozhrania inštancie entít z modulu domény, volá príslušnú inštanciu alebo triedu z domény pre vykonanie doménovej logiky, transformuje výsledok do DTO triedy a vráti spracovaný výsledok používateľovi.

```
public interface PeriodApi {

    @PostMapping(value = "/new")
    boolean create(@RequestBody PeriodDto period);

    @GetMapping(value = "/all")
    List<PeriodDto> getAll();
    ...
}

@RestController
@RequestMapping("/period")
public class PeriodController implements PeriodApi {

    @Override
    public boolean create(PeriodDto dto) {
        ...
    }

    @Override
    public List<PeriodDto> getAll() {
        ...
    }
    ...
}
```

Kód 3.3: Vzor API rozhrania s definíciou mapovaní a ich implementácii.

3.1.5 Komunikácia medzi službami

Komunikácia medzi službami tvorí najpodstatnejšiu časť systému, bez ktorej by sme nemali možnosť ho implementovať modulárne. Z implementačného hľadiska je dôležité, aby sme komunikáciu medzi službami minimalizovali a skutočne nechali REST API gateway ako jediný modul s komunikáciou s ostatnými službami. Pri vytvorení závislostí medzi službami sme taktiež zaviedli závislosť na nutnosti build-u mnohých artefaktov pri zmene v jednom module. Práve pri implementácii komunikácie sme zmenili návrh, kde všetky komunikácie sme ponechali iba v rámci REST API gateway s výnimkou pre modul riešení a testov, kde je potrebne notifikovať modul riešení s výsledkom ukončeného testu.

Samotnú komunikáciu sme implementovali pomocou rozhraní a implementáciou Feign v Spring-u. Pre umožnenie komunikácie so službou sme vytvorili proxy rozhranie s anotáciou *@FeignClient* [35] s názvom a context path cieľovej služby. Následne sme skopírovali definíciu metód s mapovaním z cieľoveho API rozhrania, ktoré sme ve-

deli následne využiť a vytvárať dopyty medzi službami. Pri štúdii a zoznámení sa s Feign klientom sme zistili, že ponúka možnosť využiť takzvané *boilerplate* rozhrania, kde pomocou dedičnosti vieme zdediť od cieľového API rozhrania jej definíciu. Takéto rozhranie vytvorili pre *UserAuthApi* v ukážke kódu 3.4.

```
import org.springframework.cloud.openfeign.FeignClient;
import sk.fmfi.listng.user.api.UserAuthApi;

@FeignClient(name = "listng-user", path = "/user")
public interface UserAuthApiProxy extends UserAuthApi {
}
```

Kód 3.4: Implementácia proxy pomocou Feign klienta pre komunikáciu so službou.

Implementácia Feign klienta v Spring-u umožňuje žiadanie adresy API prostredníctvom registra služieb ako Consul. Po mnohých pokusoch s konfiguráciami Feign klienta sme zistili, že jeho implementácia obsahuje aj load balancing na klientskej strane. Je možné vytvoriť vlastnú sémantiku pri výbere inštancií ale pre naše potreby nám postačuje predvolený *Round-robin scheduling* algoritmus.

3.1.6 Vytvorenie a odosielanie emailov

Vytváranie a následne odosielanie emailov tvorí kritickú funkcionalitu nakoľko pri vytvorení účtu sa vygeneruje náhodné heslo a je potrebné ho odoslať používateľovi. Nakoľko implementujeme nový a moderný systém, zvolili sme možnosť pre odosielanie emailov s formátovaním a dizajnom pomocou HTML predlohy.

Pre odosielanie formátovaného emailu sme vytvorili HTML predlohu obsahujúcu farebnú hlavičku a pätu s logom, medzi ktoré vložíme žiadaný obsah. Pomocou Thymeleaf engine sme umožnili vygenerovanie výsledného HTML súboru po dosadení obsahu [33] a za použitia frameworku Spring a jeho implementácie pre odosielanie emailov sme vedeli vygenerovaný HTML použiť ako telo emailu. V rámci konfigurácie aplikácie sme pridali potrebné hodnoty pre prístup k emailovému účtu s protokolom SMTP ako je host, port a prihlasovacie údaje [4].

```
public void sendTemplatedMail(String to, String subject, String body)
    throws jakarta.mail.MessagingException {
        Context context = new Context();
        context.setVariable("innerBodyInject", body);

        String process = templateEngine.process("general", context);
        MimeMessage mimeMessage = mailSender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(mimeMessage);
        helper.setSubject(subjectBase + subject);
        helper.setText(process, true);
        helper.setTo(to);

        mailSender.send(mimeMessage);
    }
```

Kód 3.5: Generovanie HTML template-ov a ich odoslanie emailom.

Samotnú implementáciu generovania a odosielania vidíme v kóde 3.5, kde skladáme hodnotu pre premennú v HTML predlohe, požiadame Thymeleaf engine o vygenerovanie HTML ako reťazec a následne ho použijeme ako telo emailu. Tak ako sme vytvorili metódu pre odosielanie emailu jednotlivcovi, existuje aj metóda pre odosielanie viacero používateľom a taktiež podporu pre odosielanie emailu bez formátovania. Príklad emailu vidíme na obrázku 3.2, kde sme odoslali informáciu a odkaz na obnovu hesla.

Obr. 3.2: Vzhľad formátovaného emailu generovaný pomocou Thymeleaf engine.

3.1.7 Modularita v REST API Gateway

REST API gateway slúži ako modul pre spracovanie dopytov od klienta, volá podľa kontextu potrebné služby a konečné odpovede zjednotí a spracovanú odpoveď odošle na klienta. Preto pri implementácii bolo dôležité dbať aj na bezpečnosť čo podrobnejšie opisujeme v sekcii 3.1.8.

V projekte gateway sme zaviedli štruktúru pre prehľadnosť a rozdelenie podľa modulov. V projekte sme vytvorili adresáre podľa kontextov:

- *common*: balík zdieľaných implementácií pre jednotlivé controller triedy a služby v rámci gateway,
- *controller*: balík adresárov pre jednotlivé moduly, ktoré obsahujú implementovaný controller s mapovaním a príslušnou service triedou vykonávajúcou dopyty na služby a potrebnú doménovú logiku,
- *dto*: reprezentačné DTO triedy, ktoré slúžia ako zaobalenie viacero entít alebo opačne zúženie entít podľa potrieb v Angular aplikácii. DTO triedy sme využili ako typ návratových hodnôt ale aj ako telo dopytov pre mapovania,
- *proxy*: Feign proxy rozhrania pre jednotlivé API modulov pre nadviazanie komunikácie,
- *security*: konfigurácia a implementácie autentifikovania dopytov od klienta.

3.1.8 Autentifikácia

V systéme sme potrebovali vytvoriť bezpečný spôsob implementovania prihlásenia a oprávnení tak, aby sa informácie nedržali na klientovi a používateľ mal skutočne prístup ku oprávneným častiam systému. Pre autentifikáciu a zaistenie oprávnenia jednotlivých častí rozhrania sme implementovali riešenie pomocou prostriedkov framework-u Spring a Spring Security [25].

Používateľ sa prihlási pomocou emailovej adresy a hesla, je mu umožnené zmeniť si heslo alebo požiadať o obnovu hesla v prípade zabudnutia. Po úspešnom prihlásení jeho autentifikácia je udržovaná pomocou JWT tokena [3] v HTTP Cookies v prehliadači klienta [28]. Pre zakomponovanie mechaniky kontroly prístupu v Spring Security sme jej funkcionalitu museli rozšíriť vlastnými implementáciami pre zakomponovanie kompatibility so systémom LIST-NG a udržovanie autentifikácie s JWT token.

Základnou funkciou Spring Security je filtrovanie premávky HTTP dopytov na službu, kde filter nepredstavuje bežné zužovanie množiny podľa logického predikátu ale vykonanie operácie pred tradičnou mechanikou spracovania dopytu a odpovede

[36]. Teda prichádzajúci dopyt prechádza najprv filtrom a po úspešnom vykonaní operácie sa spustí mechanika spracovania a volania príslušného mapovania controller-a. Vykonanie a spracovanie filtrovania sme konfigurovali v ukážke kódu 3.6. Pre umožnenie voľného prístupu bez autentifikácie pre monitoring, prihlasovanie, obnova hesla a prehľad statického obsahu sme vytvorili pole ciest `AUTH_WHITELIST`, ktoré budú ignorované filtrom. Vyčlenili sme aj cesty controller-ov, ktoré sú výlučne publikované pre dopyty od učiteľov a správcov, čím zaručíme že neoprávnený používateľ nemá umožnené vykonávať nežiadané dopyty. Vytvorili a namapovali sme službu vykonávajúcu autentifikáciu a načítanie používateľa a filter vykonávajúci operácie s JWT tokenom. Implementáciou JWT tokenov sme sa zbavili potreby registrovania *sessions*, čo nám umožnilo nastaviť dopyty ako *stateless*, teda dopyty sú bez stavové a neukladá sa žiadna reprezentácia záznamu o nich [9]. Hlavným dôvodom zavrhnutia sessions je škálovateľnosť systému, nakoľko pri viacero inštanciách REST API gateway každá z nich bude mať vlastný register sessions, s čím môže nastať problém s aktuálnou implementáciou. V budúcnosti pri zvážení a potrebe zakomponovania sessions a ich registrovanie postačuje zmeniť na 16-tom riadku správny typ *SessionCreationPolicy*, čím sa aktivuje mechanika registrovania a udržovania sessions pomocou Spring-u.

```
1     @Autowired
2     UnauthorizedEntryPoint unauthorizedEntryPoint;
3
4     @Autowired
5     AppUserDetailsService userDetailsService;
6
7     private static final String[] AUTH_WHITELIST = {
8             "/actuator/**",
9             "/auth/**",
10            "/preview/**"
11    };
12
13    @Bean
14    public SecurityFilterChain filterChain(HttpSecurity http) throws
      Exception {
15        http.cors().and().csrf().and()
16                .sessionManagement().sessionCreationPolicy(
      SessionCreationPolicy.STATELESS)
17                .and()
18                    .exceptionHandling().authenticationEntryPoint(
      unauthorizedEntryPoint)
19                .and()
20                    .authorizeHttpRequests()
21                    .requestMatchers(AUTH_WHITELIST).permitAll()
22                    .requestMatchers("/admin/**").hasAnyAuthority("
      TEACHER", "ROOT")
23                    .anyRequest().authenticated();
24
25        http.authenticationProvider(authenticationProvider());
26        http.addFilterBefore(authenticationJwtTokenFilter(),
      UsernamePasswordAuthenticationFilter.class);
27
28        return http.build();
29    }
```

Kód 3.6: Nastavenie konfigurácie Spring Security autentifikácie.

Servisná trieda *AppUserDetailsService* implementuje načítanie používateľa a overenie správnosti údajov. V kóde 3.7 vidíme implementáciu triedy a jednotlivé kroky pri validovaní používateľa. Objekt používateľa je primárne načítaný z modulu používateľov. Po nadobudnutí objektu používateľa z modulu volaním *detailsChecker.check(user)* vykonáme validáciu prihlasovacích údajov, ktorá je implementovaná framework-om.

Pre vyhnutie sa nadmerným dopytom na službu modulu sme implementovali vlastnú cache ktorá si uchováva inštancie používateľov na dobu 10 minút. V servisnej triede cache sme vytvorili metódu na vykonanie kontroly cache a vymazanie neplatných ob-

jektov. Metóde sme pridali anotáciu *@Scheduled(fixedDelay = ONE_MINUTE)*, ktorý umožní periodicky vykonať metódu bez jej implicitného volania.

```
@Service
public class AppUserDetailsService implements UserDetailsService {

    @Autowired
    private UserAuthApiProxy userAuthApiProxy;

    @Autowired
    private UserCacheService userCacheService;

    private final AccountStatusUserDetailsChecker detailsChecker = new
    AccountStatusUserDetailsChecker();

    @Override
    public AppUser loadUserByUsername(String username) throws
    UsernameNotFoundException {
        if (userCacheService.haveValidInCache(username)) {
            return userCacheService.getFromCache(username);
        }

        UserAuthDto userAuthDto = userAuthApiProxy.getAuthUserByEmail(
    username);
        ...
        final AppUser user = AppUser.build(userAuthDto);
        detailsChecker.check(user);
        userCacheService.storeUser(user);

        return user;
    }
}
```

Kód 3.7: Servisná trieda pre authentifikáciu používateľa.

JWT token obsahuje zašifrovaný JSON objekt s informáciami o používateľovi (id, meno, email a oprávnenie v systéme) a tajný klúč ktorým sa validuje integrita tokenu. Ukážka kódu triedy JwtUtils 3.8 obsahuje algoritmus využitý pri generovaní a validovaní tokenov. Pre podporu operácií sme použili knižnicu *io.jsonwebtoken*. Tajný kľuč, ktorý je súčasťou tokena, je šifrovaný pomocou algoritmu HS256. Token je poskladaný vytvorením mapy s obsahom nazvaný *claims* a následne podpísaný šifrovacím algoritmom a tajným kľúčom [3]. Pri validácii tokenu v metóde *isValid(String token)* sa pokúšame o parse-ovanie reťazca tokenu s našim tajným kľúčom, ktorá pri neúspešnom pokuse znamená problém so štruktúrou, podpisom alebo obsahom tokenu.

```
1  @Service
2  public class JwtUtils {
3
4      ...
5
6      private static final SignatureAlgorithm signatureAlgorithm =
   SignatureAlgorithm.HS256;
7      private Key signingKey;
8
9      @PostConstruct
10     public void postConstruct(){
11         byte[] apiSecretBytes = DatatypeConverter.parseBase64Binary(
   jwtSecret);
12         signingKey = new SecretKeySpec(apiSecretBytes, signatureAlgorithm
   .getJcaName());
13     }
14
15     public Cookie generateJwtCookie(AppUser userPrincipal) {
16         String jwt = generateTokenFromUser(userPrincipal);
17         Cookie cookie = new Cookie(jwtCookieName, jwt);
18         cookie.setPath("/");
19         cookie.setMaxAge((int) ONE_HOUR_SECONDS);
20         cookie.setSecure(true);
21         cookie.setHttpOnly(true);
22
23         return cookie;
24     }
25
26     public String generateTokenFromUser(AppUser user) {
27         return Jwts.builder()
28                 .setClaims(makeClaimsForUser(user))
29                 .signWith(signingKey, signatureAlgorithm)
30                 .compact();
31     }
32
```

```
33     private Claims makeClaimsForUser(AppUser user) {
34         // create map of type Claims containing users data
35     }
36
37     public boolean isValid(String token) {
38         try {
39             Jwts.parserBuilder()
40                     .setSigningKey(signingKey)
41                     .build()
42                     .parseClaimsJws(token);
43             return true;
44         } catch (...) {
45             ...
46         }
47
48         return false;
49     }
50 }
```

Kód 3.8: Úkažka servisnej triedy pre operácie s JWT tokenom.

Filtrovanie dopytov sme zabezpečili implementáciou komponentu *AuthTokenFilter*, ktorá pri filtrovaní dopytu sa pokusí o načítanie JWT tokenu z Cookie, validuje ho, získa objekt používateľa z *AppUserDetailsService* a nastaví ho ako kontext pre dopyt v službe REST API gateway. Kontext slúži ako priradenie dopytu danému používateľovi, ktorého vieme ďalej v implementácii mapovaní získať a použiť. Súčasťou filtra je obnova platnosti Cookie, ktorý v prehliadači automatický zanikne po uplynutí jeho platnosti. Teda bez nastaveného Cookie s tokenom používateľovi bude prístup zamietnutý a bude sa musieť opäť prihlásiť.

3.1.9 Základ controller-ov v gateway

Vďaka množstvu funkcionality ktorú nám ponúka nakonfigurovaná autentifikácia a zdieľaných metód v gateway module sme vyňali množstvo základných operácií do triedy *ListController*. V triede sme implementovali metódy pre získanie hodnôt z kontextu dopytu. V ukážke metódy 3.9 si načítavame používateľa, nastaveného vďaka implementácii autentifikačného filtra, a vrátime jeho identifikátor. Ďalšími implementáciami sú metódy *success(...) error(...)*, ktoré vrátia jednotný formát odpovede so správnym HTTP status kódom, dátami a poprípade chybovou hláškou. Hlavným rozšírením v budúcnosti bude pridanie podpory zaznamenávania činnosti pomocou jednoduchých metód *logActivity(...)* a enumerácií.

```java
public Long getUserId() {
    AppUser userDetails = (AppUser) SecurityContextHolder.getContext()
            .getAuthentication()
            .getPrincipal();

    return userDetails.getId();
}
```

Kód 3.9: Načítanie používateľa z kontextu dopytu.

3.1.10 Oprávnenia v kurzoch

Priradenie oprávnení v kurzoch používateľovi sa líši podľa jeho systémovej role. Študenti majú implicitne určené oprávnenia priamo v databáze. Učitelia a správcovia majú predvolene oprávnenie ako sledovatelia vo všetkých kurzoch ale na databázovej úrovni táto vlastnosť nie je zaznamenávaná, nakoľko pri vytvorení každého kurzu by bolo potrebné nastaviť oprávnenia všetkým relevantným používateľom ako aj pri vytvorení účtu učiteľa alebo správcu. Preto sme v servisnej triede *PermissionService* v gateway module implementovali algoritmus pre dodatočné priradenie oprávnení. Kroky v implementovanom algoritme metódy *getPermissionsInPeriod(Long userId, Long periodId)* sú:

1. Vyžiadanie inštancie *UserDto* pre používateľa s daným id od služby *Users*.
2. Vyžiadanie kolekcie kurzov v danom období od služby *Courses*.
3. Vytvorenie prieniku medzi oprávneniami používateľa a kurzmi v období.
4. V prípade, že používateľ je študentom vrátime ako výsledok vytvorený prienik oprávnení.
5. Vytvorenie mock oprávnení pre chýbajúce kurzy v období.

Implementovaný algoritmus je využitý na domovskej stránke, kde zobrazujeme komentár s popisom prístupu ku jednotlivým kurzom. Implementácia je antipattern nakoľko práva by mali byť jasne určené vopred a uložené v databáze. Aby sme skutočne mali všetký práva perzistované je potrebné plošne pridávať alebo upravovať používateľov, napríklad pri tvorbe nového kurzu by sme museli vyhľadať všetkých učiteľov v systéme a pridať im nové oprávnenie k novému kurzu. Zvoleným prístupom nemusíme mať predvolené hodnoty pre učiteľov perzistované a umelo im ich pridáme vrámci gateway v prípade potreby.

3.2 Používateľské rozhranie

Používateľske rozhranie je implementované ako webová aplikácia v Angular s pridanými knižnicami ako sú Angular Material, RxJs, Fort Awesome, popperjs, Bootstrap a CryptoJs. Implementovaná aplikácia funguje primárne na klientovi tak, že na serveri sa ako odpoveď používateľovi odošlú vygenerované HTML, JS a CSS súbory, kde javaskripty obsahujú transformovanú logiku z typescript tried a služieb. Následne pri potrebe dát používateľ vytvára dopyt na implementovanú gateway LIST-NG a javaskripty odpoveď spracujú a vložia dáta do HTML.

3.2.1 Štruktúra projektu

Implementované jadro aplikácie tvorí päť častí:

- *Core*: jadro logiky aplikácie, ktorá je potrebná plošne. Obsahuje implementácie ako sú API triedy pre vytváranie dopytov na gateway, rozhrania odzrkadľujúce java DTO triedy, interceptory ktoré fungujú na totožný princíp ako filter v Spring Security, asynchrónne pipe transformujúce dáta, validácia prístupu, všeobecné servisné triedy a util triedy.
- *Main*: implementácia samotného rozhrania. Je rozdelená na lazy-loaded moduly ktoré predstavujú jednotlivé možnosti v navigácii a študentské rozhranie.
- *Shared*: znovupoužiteľné komponenty a časti rozhrania. Komponenty sú implementované tak, aby boli konfigurovateľné a dali sa prispôsobiť podľa potreby.
- *Assets*: adresár rozdelený na média a zdroje pre lokalizáciu. Zdroje pre lokalizáciu obsahujú statické JSON slovníky pre slovenčinu a angličtinu.
- *Styles*: štýlové SCSS skripty, ktoré rozdeľujeme podľa typu prvkov ako sú modálne okná, formuláre, tlačidlá, premenné, filtre a iné. Rozdelené skripty tak tvoria prehľadnú štruktúru a sú spojené importom do jedného koreňového skriptu.

Štruktúra je zadefinovná tak, že všetky implementácie v časti *Core* sú dostupné v celej main aplikácii bez potreby ich importu. Preto je dôležité do Core naozaj implementovať potrebné časti pre beh aplikácie. Výnimkou sú servisné triedy. Je želané, aby mal koreň aplikácie vytvorenú jednú inštanciu každej triedy a poskytol ich komponentom pri inject-ovaní. Týmto štýlom našej implementácie zaručujeme minimálnu spotrebu pamäte, nakoľko komponenty využívajú totožnú inštanciu servisnej triedy namiesto vytvorenia novej a vlastnej.

3.2.2 Bezpečnosť a autorizácia prístupu

V aplikácii je potrebná kontrola autorizácie pri každom dopyte, aby sme v prípade chybajúcich práv presmerovali používateľa na prihlasovaciu alebo domovskú stránku a obmedzili tak jeho prístup. Základ implementácie tvorí *AuthenticationService*, v ktorom sme implementovali handle-ovanie prihlásenia, odhlásenia, presmerovania a registrovanie prístupu do časti aplikácie.

Autorizácia v systéme je implementovaná na báze HTTP Cookie a JWT tokenu. Token sa skladá z troch časti: hlavička s typom šifrovania, zakódovaný JSON objekt s claims a podpis šifrovaným tajným kľúčom, čo podrobne popisujeme v sekcii 3.1.8. Telo tokenu je kódované pomocou Base64, ktoré vieme ľahko dekódovať pomocou Angular metódy `atob()` a tak získať údaje o používateľovi. Implementačne tieto údaje sú použité pre potreby zobrazenia mena a emailu používateľa. Hodnotu oprávnenia používateľa je nebezpečné použiť, nakoľko zásahom v prehliadači si vie používateľ tieto hodnoty ľahko upraviť. Napriek tomu, že sa nepoužívajú oprávnenia je token uložený v pôvodnom zašifrovanom formáte, aby nelákal používateľov k zásahom. Token je uchovaný v úložisku session pre použitie v aplikácii ako aj vo forme Cookie, ktorá sa nedá upraviť bez poškodenia podpisu a úspešnej validácie dopytu v gateway.

Pri prihláseni so správnymi údajmi sa v systéme vykonáva táto postupnosť krokov:

1. dopyt na gateway s prihlasovacími údajmi (email a heslo),
2. overenie údajov v gateway,
3. načítanie používateľa zo služby používateľov,
4. vytvorenie šifrovaného JWT tokenu,
5. pridanie štandardného Cookie so zabezpečením proti manipulácii,
6. prijatie odpovede na strane klienta s obsahom JWT tokenu a nastaveným Cookie,
7. prehliadač zaregistruje Cookie a uchová jeho hodnotu,
8. validácia tela tokenu,
9. uloženie tela tokenu do pamäti session u klienta prostredníctvom triedy *TokenStorageService* implementujúcej správu tokena v pamäti,
10. presmerovanie klienta na domovskú stránku,
11. kontrola stráženého prístupu nastavená v konfigurácii rozcestníka v *MainRouting* pomocou *AuthGuard*, ktorý je implementovaný ako logický predikát s kontrolou správnosti prístupu.

Pre implementáciu uvedenej mechaniky sme vytvorili množstvo malých servisných tried, ktoré každým krokom validujú vstupy a v krokoch po uchovaní tokenu kontrolujeme aj platnosť tokenu. Docielili sme zabezpečenie aplikácie overovaním prístupu používateľa a zohľadnili sme možnosť manipulácie používateľom. Pri každom kroku s validáciou sme implementovali aj mechaniku pre vyvolanie chybovej hlášky, ktorá v nevyhnutnom prípade presmeruje používateľa na prihlasovaciu stránku. Taktiež dopyt pre prihlásenie na gateway pri neúspechu vráti odpoveď s 401 HTTP status kódom, ktorá je štandardom pre označenie neoprávneného dopytu.

3.2.3 Moduly a routing

Navigovanie cez používateľské rozhranie sa nazýva *routing*. Je konfigurované v routing triedach moduloch aplikácie. Ako všeobecná obálka routingu používame triedu *AppRoutingModule* kde sú nastavené cesty pre prihlasovacie okno, okná pre obnovu hesla a errorové okno. Ostatné cesty, ktoré nespĺňajú deklarované routingy sú presmerované na *MainModule* po validácii s *AuthGuard*. Touto konfiguráciou zaručujeme ako je zobrazené v ukážke 3.10, že pri každom načítaní stránky patriacej *MainModule* kontrolujeme a validujeme prístup používateľa. Úspešné presmerovanie routing ciest riadi *MainRoutingModule* s konfiguráciou pre domovskú stránku a podmoduly.

```
1  const routes: Routes = [
2      {
3          path: '',
4          component: MainComponent,
5          loadChildren: () => import('./main/main.module').then(m => m.
     MainModule),
6          canActivate: [AuthGuard],
7          canActivateChild: [AuthGuard]
8      },
9      {
10         path: 'login',
11         component: LoginComponent
12     },
13     ...
14 ];
```

Kód 3.10: Konfigurácia routing ciest.

3.2.4 Lokalizácia textov

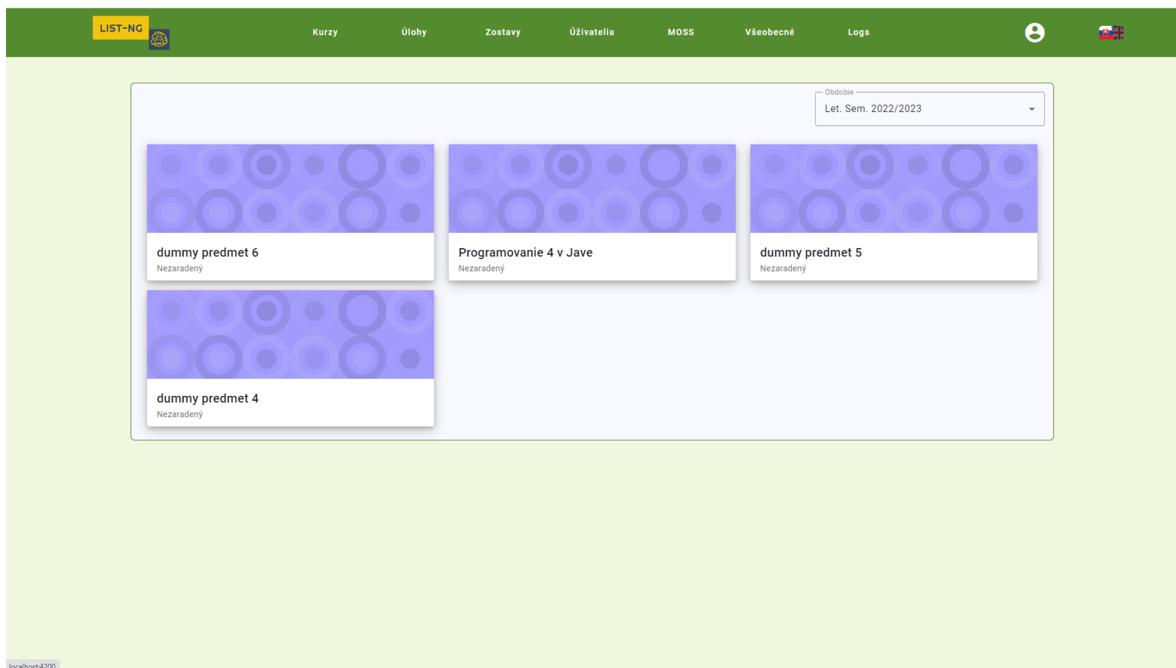
Dôležitou súčasťou aplikácie je podpora pre lokalizáciu textov a obsahu. Statické texty využité ako názvy a označenia komponentov webovej aplikácie sa nachádzajú v JSON súboroch *sk.json* a *en.json* predstavujúce slovníky. Lokalizáciu pre texty, ktoré tvoria učitelia, ako sú popisy k úlohe, podporuje opisaná trieda *MultiLangText*. Štandardne lokalizácia v Angular aplikácii je implementovaná za pomoci využitia Angular *TranslateModule* so správnou konfiguráciou [18]. Následne využitím pipe *translate* sa vykonáva asynchrónne načítanie prekladov zo slovníka. Pre podporu podobnej mechaniky sme implementovali triedu *CustomTranslatePipe* zobrazenú v ukážke kódu 3.11, ktorá rozširuje funkčnosť tradičnej *TranslatePipe*. V ukážke taktiež ukazujeme spôsob využitia vytvorenej pipe, kde v prvom prípade využívame reťazec reprezentujúci kľúč zo slovníka a v druhom prípade sme použili javaskriptový objekt typu *MultiLang*.

```
1  @Pipe({
2      name: 'translation',
3      pure: false
4  })
5  export class CustomTranslatePipe extends TranslatePipe {
6
7      ...
8
9      override transform(value: MultiLang | string, ...args: any[]): any {
10         let jsonValue;
11         let objValue = '';
12         if (typeof value === 'string') {
13             jsonValue = super.transform(value);
14         } else {
15             // spustime mechanizmus ngx-translate, aby sme vyuzili
   synchronizovany update prekladov aj pre objekty
16             jsonValue = super.transform('dummy.object');
17             objValue = this.localeService.isSlovak() ? value.SK : value.
   EN;
18         }
19
20         return typeof value === 'string' ? jsonValue : objValue;
21     }
22 }
23
24 /** HTML usage Example
25 <p> {{ 'foo.bar.static' | translation }} </p>
26 <p> {{ someMultiLangObject | translation}} </p>
```

Kód 3.11: Asynchrónna lokalizácia textov s príkladom využitia v HTML template.

3.2.5 Identita stránky

Aplikáciu používateľského rozhrania sme vyvíjali s ohľadom na nasledujúcich vývojárov a využili osvedčené postupy pre vývoj podobných aplikácii. Vytvorili sme obalenie pre jednotlivé časti rozhrania, aby sme zabezpečili jednotný vzhľad bez potreby osobitnej konfigurácie pre každé okno. Najdôležitejšia časť je telo obsahu, ktorému sme vytvorili blok s centrovaním, štýlmi a pokrýva 80% šírky okna prehliadača. Ako ukážku vidíme domovskú stránku na obrázku 3.3, kde telo je má definovaný vzhľad pre obsah. Iné obalenia sa vytvorili aj pre modálne okna, notifikácie a formuláre, ktoré vieme ľahko použiť alebo plošne zmeniť vzhľad zmenou na jednom mieste.



Obr. 3.3: Domovská stránka LIST-NG.

Interaktívne časti používateľského rozhrania sme implementovali najmä použitím konfigurovateľných komponentov z framework-u Angular Material. Ich hlavným prínosom sú existujúce implementácie konfigurovateľnej logiky a jednotné štýly komponentov. Pre naše potreby sme využili množstvo komponentov ako sú tlačidlá, navigácia, ikonky, textové políčka s automatickým dopĺňaním, základ pre stránkovanie, vstupy vo formulároch, tabuľky a mnoho iných. Pre komponenty sme vytvorili tému s troma paletami farieb, ktoré sú následne automatický dosadené v komponentoch Angular Material.

Angular Material priniesol aj principiálny problém pri potrebe zmeniť štýl komponentu. V ukážke okna 3.3 v pravom hornom rohu tela stránky sme využili komponentu `mat-select` pre rozbaľovaciu ponuku čím volíme obdobie v ktorom chceme zobraziť kurzy. Tlačidlo ponuky v sebe obsahuje veľa prázdneho miesta čo je v spore s požiadavkou pre efektívne využitie miesta na ploche. Použitý komponent sa skladá z ôsmich

```
.period-picker-simple {
    width: 20em;
    border: none;
    height: 1.5em !important;
    max-height: 1.5em !important;

    > * {
        height: 1.5em !important;
        max-height: 1.5em !important;
    }

    .mat-mdc-form-field-infix {
        padding-top: 0 !important;
        padding-bottom: 0 !important;
    }
}
```

Kód 3.12: Ukážka prepísania štýlov komponenty z Angular Material.

vrstiev HTML tagov, kvôli čomu nevieme ľahko zistiť ktorej z nich prepísať hodnoty štýlov. Totožnú komponentu v inej časti aplikácie sme úspešne zmenšili pomocou kódu v ukážke 3.12. Pre niektoré miesta je nevhodne použiť tieto komponenty, nakoľko nachádzanie správnych CSS tried je prácne.

3.2.6 Filtrovanie

Tabuľky pre správu entít v systéme mnohokrát obsahujú veľké množstvo dát, kde používateľom nebude postačovať zoradenie podľa stĺpcov na jednoduchý prehľad. Vytvorili sme štyri typy konfigurovateľných komponentov pre filtre: full-textové vyhľadanie, zadanie rozpätia hodnoty od-do, označenie pravdivosti vlastnosti pomocou trojstavového zaškrtávacieho poľa (nepriradený, áno, nie) a výber z relevantných možností. Jednotlivé filtre sú zakomponované v komponente modálneho okna, ktorý na vstupe očakáva konfiguráciu filtrov ktoré interpretuje a dynamicky vytvorí obsah s potrebnými typmi filtrovania. Príklad výsledného okna s filtrovaním vidíme na obrázku 3.4, ktorého obsah sa automatický interpretoval a vytvoril po priložení konfigurácie požadovaných polí. V ukážke kódu refimpl:filterConfig vidíme príklad konfigurácie pre filtrovanie nad tabuľkou používateľov. Modálne okno v konečnom dôsledku ovláda iba zmeny v poliach, kde po následnom potvrdení vráti zadané hodnoty a aplikovanie filtrov má na starosti volajúci komponent.

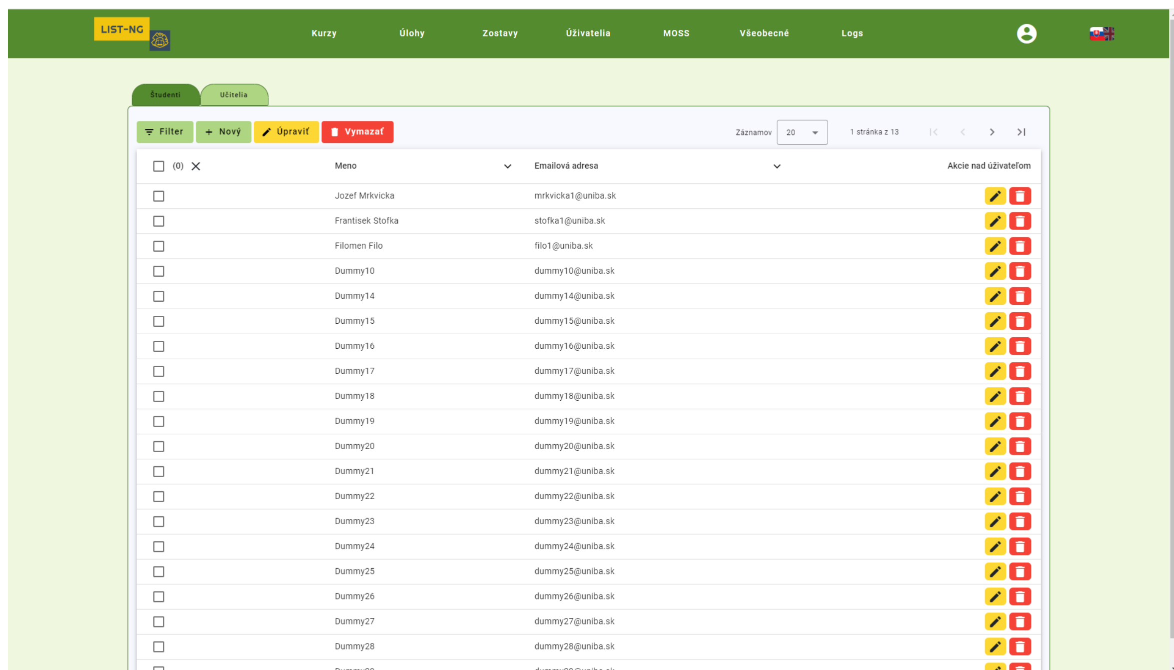
Obr. 3.4: Konfigurované modálne okno filtrovania pre okno používateľov.

```
 1
 2 this.filters = [
 3     {field: 'name', label: 'user.name', type: FilterType.FULL_TEXT, value
      : '', default: ''} as Filter,
 4     {field: 'email', label: 'user.email', type: FilterType.FULL_TEXT,
      value: '', default: ''} as Filter,
 5     {field: 'active', label: 'user.isActive', type: FilterType.BOOLEAN,
      value: undefined, default: undefined} as Filter,
 6     {field: 'course', label: 'user.course.belongs', type: FilterType.
      CHOICE, value: [], default: [], data: this.courseOptions} as Filter
 7 ];
 8 ...
 9
10 openFilter() {
11     const dialogRef = this.dialog.open(FilterModalComponent, {
12         data : { filterConfigs: this.filters }
13     });
14
15     dialogRef.afterClosed().subscribe(response => {
16         ...
17     })
18 }
```

Kód 3.13: Ukážka konfigurácie filtrov.

3.2.7 Všeobecná tabuľka pre správu

Správa entít na používateľskom rozhraní je definovaný jednotným vzhľadom v časti návrhu 2.12. Preto sme vytvorili všeobecnú komponentu `ListTableComponent` pre jednu tabuľku s ovládaním a konfiguráciou obsahu. Komponent tabuľky pre správu konfigurujeme pomocou definovania možných akcii pre entitu, hromadné akcie, potrebné stĺpce a zdroj dát. Časti definujeme pomocou ngTemplate, ktoré ako z názvu vyplýva predstavuje dosadenú predlohu pre určenú časť komponenty. Našou implementáciou teda stačí vytvoriť predlohu pre jednotlivé stĺpce v riadkoch a sekciu s tlačidlami pre vykonanie hromadných akcii. Okno zobrazené na obrázku 3.5 predstavuje správu používateľov, na ktorom používame implementáciu uvedenej komponenty `ListTableComponent`. V implementácii okna sme vytvorili predlohy pre sekciu s ovládaním nad tabuľkou, definíciu obsahu stĺpcov a použili konfiguráciu hlavičky tabuľky ako vstupný atribút. Použitý prístup vyžadoval množstvo práce, nakoľko udalosti interakcií a zmien spracováva komponent tabuľky, ktorý nemá dostup k metódam a dátam v nadradenej komponente.

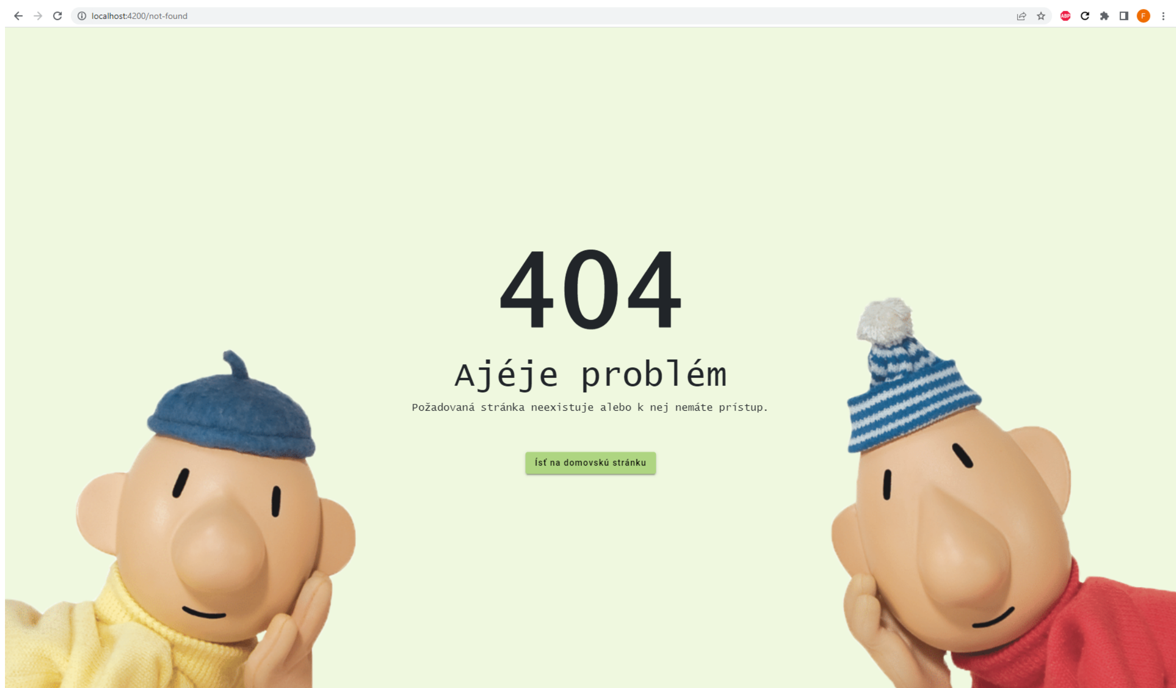


Obr. 3.5: Okno pre správu používateľov.

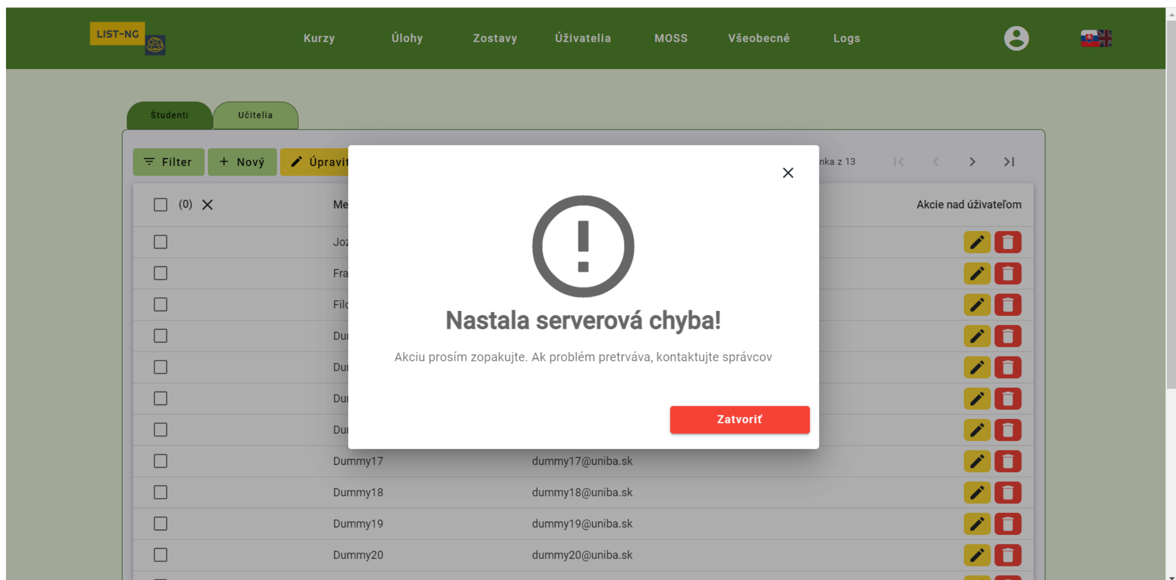
3.2.8 Ošetrovanie chýb

V aplikácii používateľského rozhrania sme implementovali ošetrenie chybných dopytov a stránok. Pri pokuse otvoriť neexistujúcu stránku alebo stránku ku ktorému používateľ nemá prístup sa automaticky otvorí okno so správou "Vyhľadávaná stránka neexistuje". Dôležitejšou súčasťou je kontroly je ošetrenie dopytov na gateway. V prí-

pade že odpoveď na dopyt nie je označený so status kódom 200 (predstavuje ÖK"), tak ho spracujeme a zobrazíme používateľovi príslušne modálne okno. Ak používateľ sa pokúsi vytvoriť dopyt na gateway, ku ktorému by nemal mať prístup tak ho informujeme o nepostačujúcich právach. Dopyt ktorý skonči chybným vyhodnotením alebo požadovaná služba pre vykonanie dopytu nie je dostupná, používateľovi zobrazujeme generické modálne okno s informáciou, že potrebná služba nie je dostupná a nech sa pokúsi kontaktovať správcu systému. Ošetrenia chybných hlášok sme implementovali tak, že ako odpoveď pre dopyt dostaneme správu na zobrazenie používateľovi a informáciu či v danom prípade chceme presmerovať používateľa na niektorú stránku.



Obr. 3.6: Chybové okno pre prípad neexistujúcej adresy alebo nepostačujúcich právach.



Obr. 3.7: Modálne okno, ktoré automaticky zobrazujeme v prípade zlyhaného dopytu.

3.3 Infraštruktúra prostredia

3.3.1 Príprava a inštalácia

Pre prostredie nasadenia systému sme vybrali Ubuntu server. Za pomoci fakultného správcu serverov sme vytvorili nový server s doménou listng.dai.fmph.uniba.sk. Pre server sme nainštalovali všetky potrebné náradia ako sú najnovšie LTS verzie Java JDK, NodeJS, binárny súbor Consul pre unix, PostreSQL a Nginx. Pre databázu sme obmedzili prístup mimo localhost-u, nakoľko vzdialený prístup do databázy nie je potrebný. Vytvorili sme SQL skripty pre založenie databázy, nastavenie používateľov pre naše služby a v jednotlivých projektoch modulov sme vytvorili skripty pre inicializovanie tabuliek pre model systému. Všetky zmeny a pokroky sme zdokumentovali v súbore *admin_notebook.log* aj s jednotlivými postupmi, ktoré budú slúžiť pre správnu prípravu prostredia v potrebe jeho obnovy. Z hľadiska bezpečnosti sme pripravili konfiguračný súbor pre uzamknutie všetkých portov okrem predvolených na SSH, HTTP a HTTPS komunikáciu.

Webový server Nginx sme zvolili primárne pre potrebu implementovania reverzného proxy, ktorú sme nakonfigurovali aby dopyty na server presmeroval na služby podľa ich cesty. Nakoľko na server bude klient posielať dopyty na Angular aplikáciu pre vyhotovenie zdrojových kódov web stránky a na REST API gateway pre načítavanie dát a vykonávanie operácii. Dopyty rozlišujeme podľa prvej hodnoty cesty za doménovým menom. Dopyt začínajúci s *'/api'* patrí REST API gateway a všetky ostatné patria Angular aplikácii. Pri vytvorení dopytu na server sa adresa analyzuje, Nginx dopyt následne presmeruje na potrebnú službu a odpoveď následne odošle naspäť klientovi.

Toto riešenie sme zvolili z bezpečnostného hľadiska a tiež kvôli problému s firewall-om používateľa, ktorý predvolene povoľuje komunikáciu so štandardnými portmi ako sú 22 (SSH), 80 (HTTP) a 443 (HTTPS). Po následnom nasadení systému a jeho využívaní je potrebné vytvoriť SSL certifikát, aby HTTPS pre prostredie a klientov fungoval správne.

Pre nasadenie aplikácie sme vytvorili bash-ove skripty pre podporu celého procesu nasadenia. Prvý skript stiahne repozitar podľa vetvy aj commit hash, spústi paralelný build projektov s unit testmi a výsledne artefakty skopíruje do určeného adresára. Ďalší vytvorený skript má na starosti validáciu integrity databázy a spustenie jednotlivých služieb. Ako možné rozšírenie do budúcna by bolo adekvátne riešenie pridať light-weight distribúciu open source repozitára pre maven artefakty ako je Nexus, v jednotlivých projektoch nakonfigurovať krok pre *maven deploy* [27] čím z vývojového prostredia ako je IntelliJ vieme zaslať arefakty na uloženie na serveri. Ako druhý návrh môžeme uvažovať nad zapracovaním nástroja pre automatizácie ako je Jenkins [15], ktorý beží na serveri a mál by potrebné automatizácie pre stiahnutie repozitára, build artefaktov, validácia, spustenie služieb a toto všetko v rámci používateľského rozhrania.

Záver

Túto prácu sme začali s uvedením motivácie a cieľov bakalárskej práce, ktoré boli vyhotovenie podrobne zdokumentovaného návrhu, implementácia základov a definovanie štruktúry systému so zámerom nahradenia existujúceho systému LIST. Zorganizovali sme mnoho konzultácii vyučujúcimi, ktorí aktívne využívajú systém LIST a ďalších učiteľov na fakulte s vlastným podobným riešením systému na odovzdanie a hodnotenie riešení. Iteratívne sme vytvárali návrhy pre architektúru systému, doménového modelu a samotnej infraštruktúry prostredia pre nasadenie.

Z implementačného hľadiska sme s využitím frameworku Spring úspešne vytvorili premyslenú štruktúru pre modulárne služby, zaviedli štandardy pre komunikáciu medzi službami, využili a rozšírili softvér pre podporu škálovania. Vyvinuli sme dve celé a dôležité doménové služby pre správu používateľov, kurzov a organizačných entít potrebné pre administráciu výučby. Implementovali sme vstupnú bránu do aplikácie, kde sme úspešne vytvorili podporu autentifikácie a autorizácie používateľa so zaužívanými štandardmi v praxi. Zohľadnili sme aj prípady, kde je možnosť výskytu zlyhania, ako je pokus o otvorenie stránky alebo volanie metód služieb bez potrebných prístupov.

Pre používateľské rozhranie sme vytvorili základnú identitu stránky a implementovali rozsiahlu časť pre autorizáciu prístupov. Hlavným cieľom bola príprava znovu použiteľných komponentov, ktoré sa budú vyskytovať v aplikácii na mnohých miestach. Ako výsledok sme docielili konfigurovateľné komponenty, ako sú formuláre pre filtrovanie dát rôzneho typu, vytvorenie tabuľky pre zobrazenie a správu entít, vytvorili verejnú službu pre lokalizáciu statických aj dynamických textov, štruktúru dialógov a iné.

Infraštruktúru prostredia sa nám nepodarilo sfunkčniť s použitím reverzného proxy na presmerovanie dopytov na jednu z dvoch modulov, ale ako výsledok sme dostali mnoho nových poznatkov a skúsenosti. Spracovali sme jednoduché skripty pre nastavenie databázy s prístupmi, vygenerovanie dát pre potrebu vývoja a primitívna automatizácia pre nasadenie systému.

Nakoľko systém je rozsiahly a bude vyvíjaný ďalšími študentmi, ďalšie kroky pre vývoj je vhodné vypracovať službu pre úlohy a zostavy úloh, riešenia študentov, modul pre správu súborov a základná verzia testovacieho systému. Vedeli by sme brať do úvahy aj rozšírenie domény úloh, kde by LIST-NG podporoval tvorbu kvízov. Modul pre

správu súborov bude jednoduchá záležitosť, ako aj testovací systém v ktorom sa použijú existujúce skripty zo systému LIST. Dôležité ďalšie kroky sú analýza a implementácia parametrov filtrovania s generickým zapracovaním a implementácia jednotného spôsobu asynchrónnej komunikácie bez potreby zavedenia ďalšej technologickej závislosti pre asynchrónne správy.

Bakalárska práca bola pre nás veľkým prínosom, nakoľko sme mali prax s riešením zadaní v rámci výučby alebo rozširovanie funkcionality už dlho existujúceho projektu implementovaný s framework-om Spring a Angular. Veľkou výzvou bolo správne zvoliť postup práce, potreba vopred zohľadniť mnoho maličkostí a najmä postavenie aplikácii a infraštruktúry od úplného základu. Najväčším prínosom z práce je rozšírenie analytického myslenia a správa servera. S určitosťou vieme povedať, že bude nás tešiť mať možnosť sa ďalej podieľať na vývoji systému.

Literatúra

[1] angular.io. Angular Docs — angular.io. https://angular.io/docs. [Accessed 02-Jun-2023].

[2] angular.io. Angular Material — angular.io. https://material.angular.io/. [Accessed 02-Jun-2023].

[3] auth0.com. JWT.IO - JSON Web Tokens Introduction — jwt.io. https://jwt.io/introduction. [Accessed 02-Jun-2023].

[4] baeldung. Guide to Spring Email — baeldung.com. https://www.baeldung.com/spring-email, 2021. [Accessed 02-Jun-2023].

[5] baeldung. Java DTO pattern — baeldung.com. https://www.baeldung.com/java-dto-pattern, 2022. [Accessed 02-Jun-2023].

[6] baeldung. Spring derived queries — baeldung.com. https://www.baeldung.com/spring-data-derived-queries, 2022. [Accessed 02-Jun-2023].

[7] DDD Community. Architektúra Microservices | Miroslav Rúčka Solution Architect v PosAm — youtube.com. https://www.youtube.com/watch?v=WZUKQ3YaeLw. [Cit. 29.5.2023].

[8] DDD Community. FILOZOFIA DOMAIN DRIVEN DESIGN | Zdeno Jašek Solution architect v PosAm — youtube.com. https://www.youtube.com/watch?v=_5jFXGhB4pc, 2019. [Cit. 30.1.2023].

[9] Lokesh Gupta. What is REST - REST API Tutorial — restfulapi.net. https://restfulapi.net/. [Accessed 02-Jun-2023].

[10] Thomas Hamilton. Unit Testing Tutorial – What is, Types & Test Example — guru99.com. https://www.guru99.com/unit-testing-guide.html, 2023. [Accessed 02-Jun-2023].

[11] HashiCorp. Consul — hashicorp.com. https://developer.hashicorp.com/consul/docs/intro, 2023. [Accessed 02-Jun-2023].

[12] Hibernate. What is Object/Relational Mapping? - Hibernate ORM — hibernate.org. https://hibernate.org/orm/what-is-an-orm/. [Accessed 02-Jun-2023].

[13] Hibernate. Your relational data. Objectively. - Hibernate ORM — hibernate.org. https://hibernate.org/orm/. [Accessed 02-Jun-2023].

[14] Javatpoint. What is modular programming - Javatpoint — https. https://www.javatpoint.com/what-is-modular-programming. [Cit. 29.5.2023].

[15] jenkins.io. Jenkins — jenkins.io. https://www.jenkins.io/. [Accessed 02-Jun-2023].

[16] Andrej Jursa. Bakalárska práca - LIST. https://dai.fmph.uniba.sk/~petrovic/th13/Jursa_list.pdf, 2013. [Accessed 02-Jun-2023].

[17] Andrej Jursa. Zdrojový kód - LIST. https:https://github.com/andrejjursa/list-lms, 2023. [Accessed 02-Jun-2023].

[18] Andreas Löw. How to translate your Angular app with ngx-translate — codeandweb.com. https://www.codeandweb.com/babeledit/tutorials/how-to-translate-your-angular-app-with-ngx-translate, 2023. [Accessed 02-Jun-2023].

[19] Olivia McGarry. What is an LMS? — learnupon.com. https://www.learnupon.com/blog/what-is-an-lms/. [Cit. 30.5.2023].

[20] PhD. Mgr. Pavel Petrovič. Materiály k predmetu Tvorba informačných systémov. https://dai.fmph.uniba.sk/courses/tvorbaIS/tis/new.html, 2022. [Cit. 21.5.2023].

[21] Microsoft. Managed kubernetes service (aks) | microsoft azure.

[22] Salvador Molina. stackoverflow.com. https://i.stack.imgur.com/JjidY.png, 2019. [Accessed 02-Jun-2023].

[23] Moodle. MoodleDocs — docs.moodle.org. https://docs.moodle.org/402/en/Main_page. [Accessed 02-Jun-2023].

[24] Vianney MORALES-ZAMORA and María Petra PAREDES-XOCHIHUA. Sanchez-juarez iván rafael. importance of the spring framework in web programming. *Journal of Computational Systems and ICTs*, pages 8–22, 2022. [Cit. 21.4.2023].

[25] Peter Mularien. *Spring Security 3.* Packt Publishing Birmingham,, England, 2010.

[26] nginx. Advanced Load Balancer, Web Server, & Reverse Proxy - NGINX — nginx.com. https://www.nginx.com/. [Accessed 02-Jun-2023].

[27] Eric Redmond. Maven; Maven Documentation — maven.apache.org. https://maven.apache.org/guides/index.html. [Accessed 02-Jun-2023].

[28] reflectoring.io. Handling Cookies with Spring Boot and Servlet API — reflectoring.io. https://reflectoring.io/spring-boot-cookies/, 2021. [Accessed 02-Jun-2023].

[29] Written by: Łukasz Ryś. Hexagonal architecture, ddd, and spring, 2022. [Cit. 29.5.2023].

[30] Alvinditya Saputra. Spring searching and filtering using JPA Specification — piinalpin.com. https://blog.piinalpin.com/2022/04/searching-and-filtering-using-jpa-specification/, 2022. [Accessed 02-Jun-2023].

[31] Jef Spaleta. Sensu | How Kubernetes works — sensu.io. https://sensu.io/blog/how-kubernetes-works. [Accessed 02-Jun-2023].

[32] Tatsuya Tamura. *Spring Boot Primer: Learn Springboot from the basics*. Tatsuya Tamura, 2021.

[33] Thymeleaf. Documentation - Thymeleaf — thymeleaf.org. https://www.thymeleaf.org/documentation.html. [Accessed 02-Jun-2023].

[34] Tutorialspoint. Hibernate - Query Language — tutorialspoint.com. https://www.tutorialspoint.com/hibernate/hibernate_query_language.htm. [Accessed 02-Jun-2023].

[35] VMware. Declarative REST Client: Feign — cloud.spring.io. https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-feign.html. [Accessed 02-Jun-2023].

[36] VMware. Spring Framework Documentation — spring.io. https://docs.spring.io/spring-framework/reference/, 2023. [Accessed 02-Jun-2023].

[37] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

Príloha A: Zdrojový kód

Zdrojový kód je zverejnený na verejne dostupnom repozitári na stránke `https://github.com/pitakfilip/listng`. V rámci projektu sú aj potrebné návody a informácie pre správnu inštaláciu a spústenie systému.