FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Automatizovaná kontrola laboratorních úloh z počítačových sítí
Student:	Tomáš Arazim
Vedoucí:	Ing. Viktor Černý
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Laboratorní úlohy kurzu Počítačové sítě kladou velmi vysoké nároky na učitele. Zvláště úlohy, ve kterých pracují studenti samostatně a v průběhu úlohy je nutné několikrát zkontrolovat, jestli student danou část pochopil a může pak pokračovat dál. Přestože jsou na hodinách přítomni dva učitelé, je velmi časově náročné kontrolovat studenty a zároveň odpovídat na jejich dotazy. Navrhněte a vytvořte systém, který by usnadnil kontrolu studentských řešení. Pokyny:

- Analyzujte aktuální laboratorní úlohy kurzu Počítačové sítě a identifikujte části, které mohou být automatizovaně kontrolovány.
- Navrhněte aplikaci, která poběží na studentských počítačích, kontroluje průchod studenta úlohou a zároveň o výsledcích informuje serverou část aplikace běžící na učitelském počítači.
- Navržené řešení bude sledovat principy softwarového návrhu a umožní snadnou rozšiřitelnost či změny.
- Realizované řešení ověřte na simulovaném laboratorním cvičení nebo po dohodě s vyučujícími přímo v ostrém provozu.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 14. ledna 2022 v Praze.

Bakalářská práce

Automatizovaná kontrola laboratorních úloh z počítačových sítí

Tomáš Arazim

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Viktor Černý
6. ledna 2022

Poděkování

Rád bych poděkoval v první řadě svému vedoucímu práce Ing. Viktorovi Černému, za pomoc při psaní této práce, za vedení obecně, za čas a konzultace. Dále bych rád poděkoval své rodině a kamarádům, za psychickou podporu a odreagování v těžkých časech.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Rovném dne 6. ledna 2022

Tomáš Arazim

Odkaz na tuto práci

ARAZIM, Tomáš. *Automatizovaná kontrola laboratorních úloh z počítačových sítí*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Tato bakalářská práce se zabývá primárně analýzou úloh síťových laboratoří, návrhem systému automatizovaných kontrol studentských výstupů a jeho následnou omezenou implementací. Implementace se zabývá kontrolou uživatelských odpovědí s pomocí definovaných maker a monitoringem studentského postupu. Výstupem je trojice aplikací – RESTful server a dva klienti.

Klíčová slova: Laboratoře FIT, Kontrola studentské práce, Scala, Java, Spring Boot, RESTful, Webová služba

Abstract

The focus of this bachelor thesis is primarily the analysis of network lab assignments, design of an automatic validation system of student output and its limited implementation. Implementation is concerned about checking user answers with the use of predefined macros and monitoring of student progress. The outputs are the application trio – RESTful server with two clients.

Keywords: FIT laboratories, Student output validation, Scala, Java, Spring Boot, RESTful, Web service

Obsah

Seznam zkratek

API – Application Programmable Interface
DAO – Data Access Object
DFS – Depth First Search
DHCP – Dynamic Host Configuration Protocol
DNS – Domain Name Service
DRY – Don’t Repeat Yourself
DTO – Data Transfer Object
HTTPS – Hypertext Transfer Protocol Secure
HTTP – Hypertext Transfer Protocol
IP – Internet Protocol
JDBC – Java Database Connectivity API
JDK – Java Development Kit
JPA – Java Persistence API
JRE – Java Runtime Environment
JSON – JavaScript Object Notation
JVM – Java Virtual Machine
MAC – Media Access Control
ORM – Object Relational Mapper
SOAP – Simple Object Access Protocol
UI – User Interface
URL – Uniform Resource Locator
YAML – YAML Ain’t Markup Language

1 Úvod

Během výuky ve vzdělávacích institucích se lze setkat s praktickou výukou. Během těchto cvičení cvičící probírají se studenty látku z praktického úhlu pohledu. Při některých cvičeních dostávají studenti zadanou práci, tato práce může být časově omezená a délka tohoto omezení se může lišit. Když student práci dokončí, obvykle si potřebuje ověřit správnost svého řešení. Způsobů jak ověřit dané řešení je spousta, nicméně mnohdy záleží na podstatě problému, který má student řešit. Někdy se dá provést kontrola svépomocí, například zobrazením správných odpovědí po napsání testu nanečisto. S rostoucí komplexitou řešeného problému klesá počet možných aplikací tohoto řešení. V takových případech může být vyžadován cvičící, aby provedl kontrolu. Tyto kontroly mohou být krátké pro zkušenou osobu, a to v řádu minut, naneštěstí tyto kontroly lze provádět v rozumném čase pouze pro nižší počty studentů.

Laboratoře BI-PSI (počítačové sítě) trpí právě problémem přetížení cvičících (na každou laboratoř jsou přítomni dva cvičící na 24 studentů). Počítačové sítě jsou předmětem této bakalářské práce. Jakmile počet přítomných studentů přeroste 12 lidí, vede to na přetížení cvičících, ti pak nestíhají v rozumném čase obstarat všechny studenty, a to jak ty, kterým látka problém nedělá, tak i ty, kteří potřebují poradit. Navíc tyto kontroly zabírají čas, který by mohli cvičící věnovat studentům, kteří projevují nadstandardní zájem o danou problematiku nebo kteří se aktivně do výuky zapojují.

Odpovědí na tento problém je automatizace, ta je již aplikována v řadě jiných předmětů, ne sice jako způsob kontroly práce v průběhu laboratoří, ale jako kontrola studentských výstupů (např. systémy Learnshell, Progtest, aj.).

Výstupem tohoto projektu je analýza požadavků systému pro kontrolu studentské práce a jeho částečná implementace, konkrétně prostředí pro zodpovídání otázek, monitorovací server s klientem.

V následujících kapitolách nejprve představím použité struktury, metody a algoritmy. Dále uvedu analytické kroky, které vedly na rozčlenění systému do výše zmíněných částí. Mimo to, představím detaily, jež vedly k vytvoření již specifických funkčních a nefunkčních požadavků. V poslední řadě popíšu svůj postup implementace, včetně vysvětlení pro daná rozhodnutí.

Cílem tedy je:

1. Analyzovat aktuální laboratorní úlohy kurzu Počítačové sítě a identifikovat části, které mohou být automaticky kontrolovány.
2. Navrhnout aplikaci, která poběží na studentských počítačích, jenž zkontroluje průchod studenta úlohou a zároveň o výsledcích informuje serverovou část aplikace běžící na učitelském počítači.
3. Navrhnout řešení tak, aby sledovalo principy softwarového návrhu a umožnilo snadnou rozšiřitelnost či změny.
4. Realizované řešení ověřit na simulovaném laboratorním cvičení nebo přímo v ostrém provozu.

2 Teoretická část

Tato kapitola vám představí metody, algoritmy a další konstrukty, které byly využity v jiných kapitolách.

2.1 Abstraktní syntaktické stromy

Stromem se v teorii grafu rozumí souvislý acyklický graf.

Grafem se pak označuje uspořádaná dvojice (*V*, *E*), kde

- *V* je konečná množina vrcholů (či uzlů).
- *E* je množina hran.

Hrana je v tomto kontextu dvouprvková podmnožina *V.*

Cesta délky *n* má *n* + 1 vrcholů spojených za sebou *n* hranami [1]

Graf je souvislý, právě tehdy, jestliže pro každé dva jeho vrcholy *u*, *v* existuje cesta z *u* do *v*. [2]

Abstraktní syntaktický strom je způsob reprezentace syntaxe jazyka s pomocí hierarchické stromové struktury. Tento strom kopíruje strukturu zdrojového kódu. Před převáděním syntaxe do této struktury je prováděn proces tokenizace (proces taktéž známý jako lexikální analýza), kdy se vybrané syntaktické výrazy transformují do abstraktní podoby [3]. Samotný význam vrcholů už pak záleží na použití, většinou je ale tento význam alespoň částečně zakódován ve vygenerovaném stromu.

2.2 Topologické řazení

Orientovaným grafem rozumíme graf, jehož hrany tvoří množina orientovaných hran, což jsou uspořádané dvojice sestavené z množiny vrcholů. [4]

Zdrojem je v orientovaném grafu označován vrchol do něhož nevede žádná vstupní hrana. [5]

Topologické řazení slouží k řazení vrcholů orientovaných grafů, tak že pro každou orientovanou hranu *uv* bude každý vrchol *u* vždy zařazen před vrchol *v*. Výsledkem je potom topologicky seřazená sekvence. Pokud se v grafu nalézá cyklus, taková sekvence neexistuje.

Toto řazení je široce využíváno například v kompilátorech, v oblasti plánování a obecně matematice. Pro realizaci tohoto řazení lze zvolit buď Kahnův algoritmus, DFS nebo řadu paralelních algoritmů. Kahnův i DFS algoritmus běží v lineárním čase.

2.2.1 Kahnův algoritmus

Kahnův algoritmus spočívá v odstraňování zdrojů z grafu. Odstraněním je vrchol přidán do výstupního seznamu. Takto postupujeme dokud v grafu existují zdroje, pokud už žádný zdroj neexistuje a v grafu nám zbývají vrcholy, topologické řazení neexistuje. [6]

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

2.2.2 Algoritmus s využitím DFS

Algoritmus DFS se používá k hloubkovému prohledávání grafů. V rámci běhu jsou označovány vrcholy do kterých algoritmus již vešel, aby nedošlo k zacyklení. Při topologickém vyhledávání se tento příznak dá využít k nalezení cyklu. Jelikož DFS nemusí díky orientovanosti grafu navštívit všechny vrcholy souvislé komponenty, je třeba jej znovu spouštět pro každý nenavštívený uzel. [7]

```
L ← Empty list that will contain the sorted nodes

while exists nodes without a permanent mark do
    select an unmarked node n
    visit(n)

function visit(node n)
    if n has a permanent mark then
        return
    if n has a temporary mark then
        stop   (not a DAG)

    mark n with a temporary mark

    for each node m with an edge from n to m do
        visit(m)

    remove temporary mark from n
    mark n with a permanent mark
    add n to head of L
```

2.3 Platformy

Tato kapitola pojednává o platformách, které je možné využít pro vybudování aplikací. Platformou rozumím softwarové a hardwarové prostředí počítače.

2.3.1 Java

Platforma Java je sada několika produktů počítačového softwaru a specifikací společnosti Sun Microsystems, která se později sloučila s Oracle Corporation. Společně poskytují systém pro vývoj aplikačního softwaru a vyvíjejí ho jako multiplatformní software [11].

Tato platforma se skládá ze dvou hlavních částí – JRE a JDK. JRE poskytuje knihovny, virtuální stroj a ostatní komponenty nezbytné pro běh appletů a aplikací napsaných v programovacím jazyku Java. Toto prostředí může být redistribuováno s aplikacemi, aby se zajistila jejich samostatnost. JDK zahrnuje JRE s vývojovými nástroji pro příkazovou řádku jako jsou kompilátory či debuggery, které jsou nezbytné, respektive užitečné, pro vývoj appletů či aplikací [12].

Jakožto platforma Java je tedy v kontextu této práce myšlen virtuální stroj, který Java využívá, a který je schopen spustit kód nezávisle architektuře nebo operačním systému (pokud je JVM na daném OS k dispozici).

2.3.2 .NET

.NET je volná open-source, managed počítačová softwarová platforma [13] pro operační systémy Windows, Linux a macOS. [14] Jedná se o cross-platform [15] následníka .NET Framework. [16] Projekt byl vydán pod MIT licencí [17], byl vytvořen společností Microsoft a je nadále spravován a vyvíjen organizací .NET Foundation.

Common Language Infrastructure (v této kapitole dále jen jako CLI) je otevřená specifikace a technický standard vyvinutý společností Microsoft by Microsoft standardizovaný jako ISO (ISO/IEC 23271)[18] a Ecma International (ECMA 335)[19]. Popisuje spustitelný kód a běhové prostředí, které umožňuje použít několika vysokoúrovňových jazyků na různých počítačových platformách, aniž by bylo potřeba upravit zdrojové kódy pro specifické architektury.

.NET se skládá z CoreCLR a CoreFX, které se dají přirovnat k CLR (Common Language Runtime) a FCL (Framework Class Library) implementacím CLI [20]. CoreCLR je zároveň virtuální stroj a kompletní běhové prostředí pro managed běh CLI programů, zároveň zahrnuje just-in-time kompilátor RyuJIT. Jedná se o implementaci virtuálního exekučního systému (VES) specifikovaného CLI. [21] CoreFX je implementací standardních knihoven specifikovaných CLI [22], zároveň ale poskytuje i další API mimo tuto specifikaci. [23]

Platformou .NET rovněž rozumím virtuální exekuční systém jak byl specifikován dle CLI. Důležitou pointou této definice je právě agnosticismus zdrojového kódu vůči platformě (pokud existuje VES pro daný OS, zdrojový kód bude nadále použitelný).

2.3.3 Native

Pod touto platformou mám na mysli libovolný operační systém na libovolné architektuře. Kód pro tuto platformu může být napsán na míru, tedy přenos kódu na jiný systém může být proveditelný, ale nemusí.

2.4 Programovací jazyky

Pro tuto práci přišlo v úvahu několik programovacích jazyků. Něco málo o těchto jazycích se můžete dočíst v této kapitole.

2.4.1 Java

Java je vysokoúrovňový objektově orientovaný programovací jazyk. Je navržen tak, aby mohl programátor napsat jeden kód a spustit ho kdekoli [24]. Tím je myšleno že zkompilovaný Java kód může běžet na libovolné platformě, která podporuje Javu bez potřeby rekompilace. [25] Java aplikace jsou typicky kompilovány do bytekódu, který může běžet na kterékoli Java virtual machine (JVM), tedy bez ohledu na počítačovou architekturu. Syntaxe je podobná jazykům C a C++, ale poskytuje méně nízkoúrovňových funkcí. Java podporuje reflexi, což není vlastnost typicky přítomná u tradičních počítačových jazyků. [26]

2.4.2 Scala

Scala je moderní multiparadigmatický programovací jazyk designovaný za účelem stručně, elegantně a typově bezpečně vyjádřit běžné návrhové vzory. Bezproblémově integruje vlastnosti objektově orientovaných i funkcionálních jazyků.

Scala je designovaná tak, aby spolupracovala s populárním Java Runtime Environment (JRE). Zejména interakce s mainstreamovou objektově orientovanou Javou je téměř nepostřehnutelná. Novější vlastnosti Javy jako SAMs[1], lambda funkce, anotace a generické metody/typy mají ve Scale přímé analogy.

Ty vlastnosti, které k sobě nemají Java analog, jako jsou výchozí nebo pojmenované parametry se kompilují tak blízko k Javě jak to jen rozumně možné. Scala využívá stejný kompilační model (rozdílná kompilace, dynamické načítání tříd), jelikož Java poskytuje přístup k tisícům existujícím vysoce kvalitním knihovnám. [27]

2.4.3 C#

C# je moderní objektově orientovaný, typově bezpečný programovací jazyk. S jeho pomocí lze vytvářet mnoho typů bezpečných a robustních aplikací které běží pod platformou .NET. Má své kořeny v rodině jazyků C, čímž je podobný jazykům C, C++, Java nebo Javascript.

Mezi vlastnosti C# patří garbage collection, který automaticky sbírá paměť obsazenou nedostupnými objekty. Nulovatelné typy, jež jsou ochráněny proti proměnným, které nereferují k alokovaným objektům. Lambda výrazy, díky kterým jazyk podporuje techniky funkcionálního programování. Language Integrated Query (LINQ) zajišťuje jednotný vzor pro práci s daty z libovolného zdroje. Dále podporuje definici jak referenčních i hodnotových datových typů. V neposlední řadě má podporu pro generické metody a typy. [28]

2.4.4 C++

C++ je multiparadigmatický programovací jazyk vytvořený Bjarnem Stroustrupem jako rozšíření programovacího jazyka C či „C s třídami“. Jazyk se skrze čas rozrostl a moderní C++ má objektově orientované, generické a funkcionální vlastnosti v doplňku s nízkoúrovňovou manipulací paměti. Je většinou implementován jako kompilovaný jazyk a mnozí výrobci jej poskytují pro svoji platformu. [29]

1 Single Abstract Method rozhraní

Design C++ je orientován k programování systémů (včetně integrovaných). Hlavním faktorem pro jeho použití je jeho rychlost, efektivita a flexibilita. Je ale používán i jinde, kde je potřeba dobrá infrastruktura nebo kde jsou velmi omezené systémové zdroje. [30] To zahrnuje desktopové aplikace, videohry, servery nebo jiné aplikace, kde je výkon nezbytný. [31]

C++ je od počátku standardizován, nejnovější verze, C++20, je standardizována pod ISO/IEC 14882:2020. [32]

Pro účely této práce bych klasifikoval tento jazyk jako jazyk běžící pod platformou native.

2.5 Metody síťové komunikace

V této kapitole shrnu metody určené pro síťovou komunikaci. Jedná se soubor architektur, protokolů a jiných souvisejících pojmů, které bylo potřeba vzít v potaz během výběru.

2.5.1 SOAP

SOAP Verze 1.2 je lehký protokol určený pro výměnu strukturovaných informací v decentralizovaném distribuovaném prostředí. Používá technologií XML pro definici rozšiřitelného frameworku zpráv, poskytujíc konstrukt zprávy, který může být vyměňován skrze mnoho různých protokolů. Tento framework byl designován tak, aby nebyl závislý na konkrétním programovacím modelu nebo na jiných implementačně specifických sémantikách.

Dva hlavní cíle SOAPu jsou jednoduchost a rozšiřitelnost. SOAP se pokouší tyto cíle splnit vynecháním vlastností, které lze nalézt v mnohých distribuovaných systémech. Mezi tyto vynechané vlastnosti patří například: spolehlivost, zabezpečení, korelace, směrování a "Message Exchange Patterns" (MEPs). [33]

2.5.2 REST

REST je client-server architektonický styl. Mezi hlavní omezení, které stanovuje jsou:

- Bezstavovost – tedy každý požadavek musí poskytnout nezbytné informace pro jeho zpracování, nemůže tedy na serveru uchovávat žádného uloženého kontextu.
- Ukládání do mezipaměti – toto omezení specifikuje, že každá odpověď musí explicitně uvádět, zda je jí možné uložit do mezipaměti.
- Jednotné rozhraní – aby mohlo být splněno toto omezení, je třeba aby platilo že:
  - Zdroje jsou identifikované
  - Manipulace zdrojů se řídí reprezentacemi
  - Zprávy jsou sebe-popisující
  - Hypermedia jsou hnacím motorem stavu
- Vícevrstvá architektura

[34]

Nutno upozornit na běžnou mylnou představu, a to tu, že REST vyžaduje využití HTTP. Není to pravdou, neboť REST je pouze specifikace, HTTP je tedy pouze implementací RESTu. Pokud jsou splněny předchozí body, pak lze jakoukoli službu (s jakýmkoli protokolem) považovat za RESTovou.

2.5.2.1 N+1 Problém

Běžně se mluví o problému N+1 v kontextu ORMs. Tento termín znamená, že když takový mapper načítá 1 databázovou entitu, musí načíst i všech ostatních *n* objektů, na které entita odkazuje. Jelikož je většinou líné načítání entit v těchto nástrojích ve výchozím stavu vypnuté, musí se načíst všech N+1 objektů, přestože je všechny nepotřebujeme.

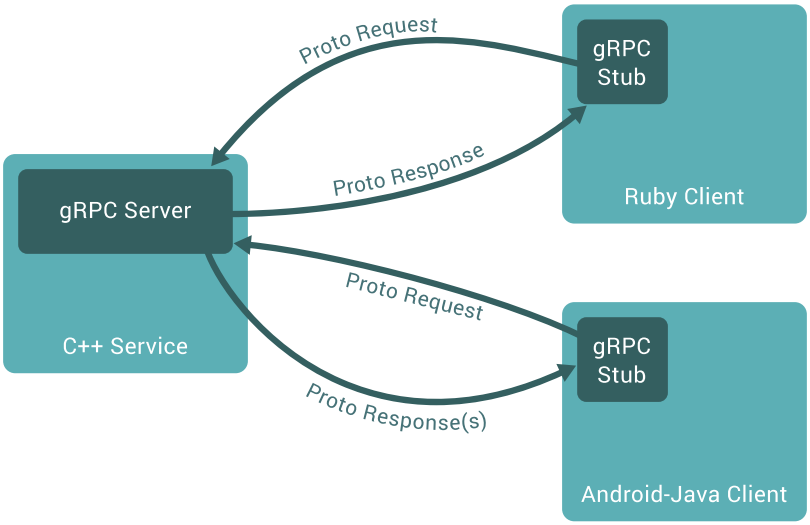
U REST rozhraní je tento problém analogický, ne vždy rozhraní vrací přesně ta data, která klient vyžaduje. Pokud nám rozhraní vrátí více informací, než potřebujeme, pak jsme degradovali na předchozí problém. V případě, že nám rozhraní nedodá všechny potřebné informace, můžeme být nuceni až n-krát volat koncové body rozhraní, abychom získali všechny údaje. [41]

2.5.3 gRPC

> Remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel. [35]

V gRPC, klientská aplikace může přímo zavolat metodu na serverové aplikaci, na jiném stroji, tak jako kdyby se jednalo o lokální objekt. Tím činí vytváření distribuovaných aplikací a služeb jednoduchým. Jako u mnohých jiných RPC systémů, gRPC je založeno kolem ideje definice služeb, za specifikování metod, které mohou být zavolány vzdáleně s jejich parametry a návratovými typy. Na straně serveru, server implementuje toto rozhraní a spustí gRPC server, aby mohl zpracovávat klientské volání. Na straně klienta existuje stub v některých jazycích je přímo označován jako klient který poskytuje stejné metody jako server. [36]

Komunikace gRPC ilustrována na obrázku č. 1.



Obrázek 1: Komunikace mezi rozhraními gRPC

2.5.4 Falcor

Falcor je inovativní datová platforma, na které jsou založeny UI Netflixu. Funguje na principu, agregace všech datových zdrojů do jednoho virtuálního JSON grafu. Klienti si při dotazech vybírají, kterou část dat z této virtuální struktury získat. Přestože se nejedná o RESTový middleware, Falcor zachovává, několik RESTových principů, konkrétně bezstavovost, ukládání do mezipaměti a vícevrstvou architekturu. [38]

2.5.5 GraphQL

GraphQL je dotazovací jazyk pro API, a serverové běhové prostředí pro spouštění dotazů s typovým systémem, který lze definovat pro konkrétní data. GraphQL není vázaný na žádnou specifickou databázi nebo databázový systém a je místo toho podporovaný existujícím kódem a daty.

Služba GraphQL se vytváří definicí typů a polí pro tyto typy. Poté je možné poskytnou funkce pro každé typ nebo pole. Poté co služba běží, může přijímat GraphQL dotazy k validaci a provedení. Služba nejprve zkontroluje jestli dotaz referuje jen k definovaným typům a polím, načež spustí poskytnuté funkce za účelem vyprodukování výsledku. [39]

3 Analýza

V této kapitole se dočtete, které kroky byly podniknuty k zmapování funkčních a nefunkčních požadavků. Jelikož žádné předchozí automatické řešení neexistuje, bylo potřeba prozkoumat existující laboratorní úlohy, odhalit tak, co je jejich cílem a jaké nároky kladou na studenta. Na základě těchto poznatků byla s pomocí konzultací vytvořena řada požadavků. Dále pak kapitola popisuje minimální životaschopný produkt. V poslední části pak lze najít požadavky s případy užití.

3.1 Laboratorní úlohy

Invariant: *Bez ohledu na to jak jednoduchý je úkol zadán, pokud je v takovém kontextu možnost selhat, tam dříve či později někdo selže*.

Předchozí věta není zcela exaktní, nazývat jí tedy invariantem je lehce zavádějící. Každopádně je potřeba, minimálně ze sociologického hlediska, vzít v potaz lidskou přirozenost. Zároveň také komplexitu našeho prostředí. Kdokoli může mít někdy špatný den, málo spánku, atd. A právě během takových dnů může toto selhání přijít velmi rychle, ať už je důvod jeden nebo jich je více. Může se jednat i o vzácné případy, nicméně vzácnosti nelze opomíjet.

Tento invariant byl podstatný při analýze cvičení PSI, pro tu jsem využil speciální znění: *Kdekoli je v zadání možnost selhat, tam nějaký student dříve či později selže.* Pravdivosti přikládají i zkušenosti cvičících, kdy se každý rok setkávají s alespoň jedním případem unikátního problému.

Pro co nejpřesnější analýzu bylo třeba projít všechna cvičení BI-PSI, nicméně ve většině případů se jedná o typově podobná zadání. Na základě invariantu jsem sestavil řadu zlomových bodů, tyto body označují, kde v zadání může student selhat. Tyto body potom musí být ošetřeny s pomocí testovacího prostředí. Druhé cvičení zcela stačí pro ilustraci opakujících-se hlavních nároků.

> Úkol: *Zapojte a nakonfigurujte segment lokální sítě*
> 1. Detaily o učebně naleznete zde: Síťová laboratoř
> 2. Zapojte kabely podle obrázku.
> 3. Na PC nastavte IP adresu na rozhraní eth1 podle obrázku.
> 4. Připojte se k Cisco směrovači. → manuál
> 5. Prostudujte si režimy směrovače. → manuál
> 6. Na Cisco směrovači identifikujte rozhraní, se kterým budete pracovat (jeho označení je na obrázku). → manuál (Přečtěte si sekci "Typy síťových rozhraní" a následující sekci "Informace o rozhraní")
> 7. Na Cisco směrovači nakonfigurujte IP adresu podle obrázku. → manuál
> 8. Utilitou ping ověřte funkčnost spojení mezi PC a směrovačem. → návod pro Cisco

Body selhání:

1: Špatně zapojené kabely
2: Na rozhraní je nastavena špatná IP adresa (směrovač | přepínač | PC).
3: Adresa je nastavena na špatném rozhraní (směrovač | přepínač | PC).
4: Připojení na špatné rozhraní (např. na přepínač místo směrovače).
5: Zapomene na správný režim aktivního prvku.
6: Odpověď na ping se vrátí ze špatného rozhraní.

Už při tomto úkolu je vidět nedostatek kontroly svépomocí. Kontrola pomocí utility ping se dá rozdělit na tři následující scénáře:

- Nejhorší scénář - Utilita ping projde, nicméně student i přesto selhal podle bodu 2 (například širší maska sítě na směrovači).
- Běžný scénář - Utilita ping selže na některém z předchozích bodů či na několika jejich kombinacích.
- Nejlepší scénář - Utilita ping projde a v konfiguraci neexistuje jediný bod selhání.

Zde nám nejhorší scénář ukazuje problém, který by student (jakožto nezkušená osoba) nebyl schopen odhalit.

> Úkol: *Pomocí aplikace Wireshark na PC zachyťte pakety ARP protokolu.*
> 1. Spusťte záznam komunikace na rozhraní *eth1* pomocí aplikace Wireshark.
> 2. Vypište si aktuální ARP tabulku. → manuál
> 3. Vynuťte si použití ARP protokolu:
>   - Pokud je v ARP tabulce přítomen záznam o IP adrese 192.168.1.2, smažte jej. → manuál
>   - Proveďte ping na IP adresu 192.168.1.2.
>   - Předchozí ping opakovaně přerušte a spusťte znovu.
> 4. Ukončete záznam komunikace.
> 5. Opět vypište aktuální ARP tabulku.
> 6. Analyzujte komunikaci ARP protokolu a odpovězte na následující dotazy:
>   - Jaká je cílová MAC adresa ARP dotazu a proč?
>   - Jaké jsou MAC adresy odpovědí?
>   - Kdy je volán protokol ARP nebo kdy se přidává záznam do ARP tabulky?
> 7. Svá zjištění sdělte vyučujícímu.

Body selhání:
1: Chybná identifikace polí v programu Wireshark.
2: Chybné odpovědi.

Pro toto zadání je vhodné upravit otázky tak, aby se daly jednodušeji ohodnocovat. Bod 6.1 lze rozdělit na dvě otázky, kterou každou lze ohodnotit. První část pak lze ohodnotit automaticky, zatímco důvod může ohodnotit cvičící – už zde je práce ulehčena. U bodu 6.2 stačí přidat do zadání formát a hned lze vyhodnocovat automaticky. Poslední podbod poté lze buď upravit s pomocí multichoice pole, které lze ohodnotit automaticky, nebo se musíme spolehnout na manuální kontrolu.

I v případě manuální kontroly, se dá proces urychlit. Pokud máme přístup k mnoha informacím, při vhodném výběru můžeme provádět vícenásobnou kontrolu bez přepínání kontextu. Díky centralizovaného monitoringu by šlo poskytnout potřebná data libovolnému klientu, při čemž ten poté dokáže data prezentovat přesně k tomuto účelu.

Úkol: *Analyzujte protokol DHCP.*

1. Na PC spusťte předkonfigurovaný DHCP server. → manuál Základní vlastnosti předkonfigurovaného DHCP serveru:
   - Nastavuje na *eth1* IP adresu 192.168.1.1/24.
   - Přiděluje IP adresy z rozsahu 192.168.1.100 až 192.168.1.150.
   - Defaultní bránu dává 192.168.1.1.
   - Jako DNS posílá adresu 10.3.44.12 (DNS pro síťovou laboratoř).
2. Spusťte záznam komunikace na rozhraní *eth1*.
3. Na Cisco směrovači nastavte rozhraní jako DHCP klienta. → manuál
4. Vyčkejte až Cisco směrovač obdrží IP adresu (tedy az zacne fungovat ping na 192.168.1.1). → manuál
5. Ukončete záznam komunikace.
6. Analyzujte DHCP protokol (filtr *bootp*) a odpovězte na následující dotazy:
   - Kolik zpráv si musí DHCP klient a server vyměnit pro první získání IP adresy klientem?
   - Jaké jsou cílové MAC adresy všech zpráv?
   - Jaké jsou IP adresy všech zpráv?
   - Jaké informace obsahuje DHCP offer zpráva?

Body selhání:

1: Chybné odpovědi.

Kontrola všech otázek z tohoto segmentu se dá plně automatizovat. Poslední otázka 6. bodu pak vyžaduje buď úpravu s využitím zaškrtávacích polí nebo postačí regulární výraz.

Úkol: *Nakonfigurujte vlastní DHCP server.*

1. Na PC zastavte DHCP server, který běží na rozhraní eth1: → manuál
2. Na PC nastavte na rozhraní *eth2* IP adresu 172.16.1.1/24. Nezapomeňte přepojit kabely z eth1 na eth2!!!
3. Upravte aktuální konfiguraci DHCP serveru: → manuál
   - Bude vydávat IP adresy v rozsahu 172.16.1.100 - 172.16.1.200.
   - Defaultní brána (routers) je 172.16.1.1.
   - Doména bude „psi.fit.cvut.cz“.
   - Projděte zbytek konfiguračního souboru a upravte ostatní IP adresy tak, aby korespondovaly se zadaným rozsahem.
4. Připojte k vašemu novému DHCP serveru Cisco směrovač a pomocí Wiresharku ověřte funkčnost vaší konfigurace.

Body selhání:

1: Zapomenutí na vypnutí běžícího DHCP.
2: Nastavení nové IP adresy na špatné rozhraní.
3: Špatně zadaná nová IP adresa.
4: Nepřepojení kabelů.
5: Špatný rozsah adres.
6: Špatná výchozí brána.
7: Neplatná doména.
8: Špatná doména.
9: Chybný formát DHCP konfigurace.
10: Chybné DNS.

11: Chybná broadcast adresa.
12: Spuštění DHCP na špatném rozhraní.
13: Směrovač má nastavenou pevnou adresu na připojeném rozhraní.

Tuto sekci je možné plně automatizovat pomocí:

- porovnání nakonfigurované IP adresy a masky
- zparsování konfiguračního souboru DHCP serveru
- kontroly položek DHCP konfiguračního souboru
- nahlédnout do leases – měla by být alespoň jedna aktivní
- pingu na tuto leased IP
- porovnání MAC lease s MAC hostitele (měly by být různé)

V dalších laboratořích se často opakují typově podobné problémy, tedy jejich automatizace bude zajištěna za pomocí výše navržených řešení. Navíc je potom: ping pro IPv6, kontrola spuštěného směrování, směrovací tabulky a vyžádání HTTP GET požadavku z učitelského počítače.

3.1.1 Sběr požadavků

Na základě laboratorních úloh lze vytvořit řadu funkčních požadavků pro software, který by byl schopen vypomoct s výukou laboratoří. Tato analýza by ovšem nebyla kompletní bez prokonzultování závěrů s vedoucím práce.

Podle metod softwarového inženýrství bych zároveň měl prodiskutovat funkcionalitu i s koncovými uživateli, tedy studenty a cvičícími. Díky této metodě pak lze získat další případy využití, případně postřehy, na které by se jinak mohlo zapomenout. Zástupce cvičících je obsažen ve vedoucím práce. Za zástupce studentů nikdo konkrétní zvolen nebyl, jelikož studenti mohou v tomto případě poskytnout pouze zpětnou vazbu/návrhy k uživatelskému rozhraní aplikace. Jelikož UI není moje přednost, lze vzít v potaz velmi důkladné oddělení funkcionalit, aby případný pokračující projekt mohl frontend nahradit.

Během konzultací vzešla pointa, která se týkala cíle, podle kterého by studenti měli tento nástroj používat jakožto nápovědu, ale zároveň jej nevyužívat jako náhradu dostupných nástrojů. Proto by měl mít cvičící možnost nadefinovat nápovědu pro případ neúspěchu, taková nápověda potom může být pouze postrčení k identifikaci konkrétního problému bez udání specifického bodu selhání.

Účely aplikace lze rozdělit do následujících modulů:

- Prostředí pro testování korektnosti laboratorní úlohy.
- Prostředí pro testování získaných znalostí.
- Software schopný monitorovat postup.

Monitorovací software se pak dá rozdělit na dvě části: klientskou a serverovou. Serverová strana data shromáždí, klientská část sesbíraná data zobrazí.

Během manuální kontroly, student může ukázat svůj postup a výsledky. Za účelem prohloubení studentovy znalosti mohou být úkoly uzpůsobené tomu, kde student v laboratoři sedí (cílem úloh nebývá memorizace). Pokud bychom ale prováděli kontrolu automatickou, je potřeba vzít v potaz tuto unikátnost. S touto eventualitou je potřeba nakládat už od zadání, je třeba nadefinovat správnou konfiguraci. Konkrétně – u otázky třeba chceme znát správnou MAC adresu nebo u testů konkrétní IP adresu na kterou provést ping. Pro tyto příležitosti jsou potřeba makra, která nahradí klíčové výrazy právě korektní hodnotou.

Nejpodstatnější makra jsou IP a MAC adresa hostitele; MAC adresa počítače s konkrétní IP adresou a číslo počítače. Poslední jmenované makro nám pomůže v příkazech, ve kterých chceme proměnlivé zadání. Dále by bylo vhodné, aby se dala tyto makra kombinovat.

Prostředí pro testování znalostí mělo umožnit definování otázky, typu odpovědi a způsob ověření správnosti. Aby se snížila obtížnost nasazení, řešení by mělo být schopné tyto definice distribuovat. Dále by pak měla být umožněna manuální oprava odeslaných odpovědí v případě, kdy neexistuje rozumný způsob automatické opravy nebo v případě nečekané chyby.

Pro studenta by mělo toto prostředí zobrazit otázku, umožnit mu odpovědět na danou otázku včetně získání zpětné vazby ve formě automatické či manuální opravy.

Prostředí pro testování korektnosti by mělo umožňovat vytváření testovacích scénářů, včetně nápovědy, co udělat když test neprochází. Stejně jako při testování znalostí, i zde by se hodilo mít systém distribuce.

Testovací prostředí by mělo disponovat příkazy s následujícím chováním:

- Zkontroluje jestli je na rozhraní *iface* stanovená IP adresa *ip* s maskou *netmask.*
- Zkontroluje buď jestli je výchozí brána nastavená na adresu *ip*, nebo není definovaná.
- Provede příkaz ping na adresu *ip*, vrací false pokud se nevrátí odpověď.
- Provede zparsování konfiguračního souboru DHCP, při selhání vrací false.
- Ověří, že je DHCP nastaveno dle definice příkazu.
- Vrátí číslo aktivních DHCP leases.
- Vrátí MAC adresu.
- Zjistí zda směrovací tabulka obsahuje routu stanovenou dle filtru.
- Spustí příkaz nativní cílovému systému a vrátí jeho výstup.

Pro každý příkaz by mělo jít určit doménové jméno či IP zařízení, na kterém by měl být příkaz proveden, ve výchozím stavu bude proveden na lokálním stroji. Pokud bude využit vzdálený cílový bod, je potřeba poskytnout validní řetězec označující operační systém (tuto informaci nelze spolehlivě zjistit automaticky).

Pokud by měl být příkaz proveden na jiném stroji (požadovaná adresa na stroji není nikde nastavená), program by měl definovat chování, které ho k danému stroji dostanou (tedy třeba připojení SSH a chování každého z příkazů na vzdálené SSH konzoli).

Rozhraní by pro definice testů mělo fungovat jako např. zjednodušené Gitlab CI s výše definovanými funkcemi a dalšími operacemi jako například podmíněnými výrazy.

Studentovi by tato část měla umožnit otestovat svojí aktuální konfiguraci, zvolit který test, či skupinu spustit a zobrazit nápovědu.

Jelikož se v laboratorních úlohách některé body opakují, aby se předešlo zbytečným duplicitním definicím, bylo by vhodné definice skládat do konkrétních bloků. Tyto bloky budou strukturou podobné laboratorním úkolům, proto je třeba, aby bylo možné definovat jejich pořadí, či závislost. Neboť zatímco některé z nich lze vyřešit kdykoli, jiné mohou záviset na dokončení předchozího úkolu.

Aby byla co nejjednodušší organizace do bloků, definice by měly zachovávat adresářovou strukturu.

Monitorovací server by měl kontrolovat stav studentů. Shromažďovat by pak měl:

- počet neprocházejících testů;
- údaje o tom, které testy neprochází a komu;
- které otázky studenti zodpověděli, společně s jejich vyhodnocením;
- konkrétní odpovědi zaslané studenty.

Během shromažďování požadavků přišla řada i na myšlenku případné gamifikace, nebo obecně rozšíření API. Shromažďovaná data by tak bylo možné dále využít. Jelikož by se mělo jednat o server, lze do něj zakomponovat distribuci testovacích scénářů a otázek.

3.1.2 Vyhodnocení

Sestavená řada funkčních požadavků, vyplývající z analýzy laboratoří, jako samotná nestačí. Je třeba pamatovat na splnění podmínky snadných změn. Když se pohybujeme v oblasti síťových technologií, cvičení se mohou rok od roku měnit díky postupujícím technologickým trendům. Samozřejmě nemusí jít ani o technologickou změnu, změnu struktury předmětu, či laboratoří samotných, stačí jenom jednoduchá úprava zadání, díky které se mohou požadavky změnit.

Změny u otázek jsou téměř neproblematické. V kapitole sběr požadavků byl dokonce navržen způsob. S dobrým designem aplikace by mělo jít takovéto situace jednoduše upravit (např. změnou konfiguračního souboru). Kde už potíže nastat mohou je testování samotného studentského postupu. Takové změny jsou, na rozdíl od otázek, vázané na konkrétní infrastrukturu, a to ať už hardwarovou nebo softwarovou. Navržený software je ve většině případů (při volbě vhodného programovacího jazyka) možné interpretovat jiným interpretem nebo překompilovat a získat tak zpět funkcionalitu lokálního testování. Když se ale jedná o testování přes síť, tam už se jde o překážku, kterou kompilace nevyřeší. Při této úloze je testování přes síť elementem, se kterým je potřeba nejen počítat, ale i s ním pracovat – některé kontroly mohou vyžadovat přístup na aktivní síťový prvek.

Při vývoji softwaru se hojně využívá metoda testovacích frameworků. Stejně jako se dají otestovat funkcionality tříd, nebo rozhraní, tak se dají otestovat i námi vyžadované scénáře. Krásou tohoto přístupu jsou změny pouze testovacího kódu, nikoli kódu frameworku. Scénáře uvnitř laboratoří, jejich struktura atp., to jsou vše ve skutečnosti jen komponenty uživatelského testovacího kódu. Vše co framework tedy musí zprostředkovat je komunikace. Pod nejjednodušším případem si můžeme představit aplikaci, co pouze přenese příkazy (nativní pro operační systém na konkrétním aktivním prvku) na onen cílový prvek přes vhodný síťový protokol, provede tento příkaz a vrátí zpět boolovskou hodnotu. Tuto funkcionalitu je možné obohatit i o další vylepšení, která mohou práci s takovým frameworkem usnadnit.

3.2 Minimální životaschopný produkt

Přestože z analýzy vyplývá poměrně široká škála požadavků, je jako minimum požadováno prostředí pro kontrolu odpovědí a jejich monitoring. Kompletní realizace všech požadavků by nebyla v časovém intervalu 4 měsíců proveditelná.

3.2.1 Prostředí pro kontrolu odpovědí

Díky podstatné redukci rozsahu lze upozornit na to, že se nyní z části funkcionálně překrývá s na fakultě vzniklým a využívaným systémem Marast; nebo se systémem Moodle. Nicméně, použití libovolného z těchto dvou systémů je mírně nevhodné, v první řadě ani jeden z nich nemá makra, která jsou klíčová v případě předmětu PSI, kdy se každého studenta týká jiný počítač. Plánované změny otázek by měly prověřit studentovo pochopení, tedy odpovědi na dané otázky tak mohou záviset na pozici studenta v učebně.

Další nevýhodou systému je jeho centrálnost, studenti během cvičení nemusí mít přístup k síti včetně vnitřní sítě fakulty. Tento problém by se dal zajisté překonat nasazením systému na počítač v učebně, nicméně i to by zahrnovalo nutnost připojit se do lokální sítě. Studenti za normálních podmínek vytvářejí vlastní síťové segmenty a není po nich, vyjma jediného cvičení, vyžadováno připojování do sítě učebny. Při zadání takového požadavku by došlo ke vzniku dalších bodů selhání.

I tento problém by šel obejít např. vytvořením skriptu, který by připojení zajistil, nicméně zde už tvoříme třetí řešení, které by pořád nemělo vyřešený první nedostatek.

V poslední řadě, výše zmíněné systémy nelze rozšiřovat ani ve směru testování, ani ve směru API, s pomocí kterého by šlo laboratoře „gamifikovat“ (zmíněné systémy se specializují v jiném směru). „Gamifikace“ sice není požadavkem akutním, nicméně je třeba postupovat tak, aby tento požadavek v budoucnosti mohl systém zprostředkovat.

3.2.2 Funkční požadavky

F1: Programu bude možné definovat otázky, typ odpovědi a způsob jakým zkontrolovat jejich správnost.

F2: Tyto definice musí jít načíst a to buď skrze síť nebo z lokálního konfiguračního souboru.

F3: Definice mohou obsahovat takzvaná makra, ty budou v době běhu programu nahrazovány konkrétními hodnotami, mezi nezbytná makra patří:

- **{host_ip:[iface]}** – zjistí IPv4 adresu hostitele na specifickém rozhraní (nebo výchozím)
- **{host_mac:[iface]}** – zjistí MAC adresu hostitele na specifickém rozhraní (nebo výchozím)
- **{host_ip_v6:[iface]}** – zjistí IPv6 adresu hostitele na specifickém rozhraní (nebo výchozím)
- **{dest_mac:[ip]} –** zjistí MAC adresu počítače se specifickou IP adresou, zde stačí když poskytne adresu v lokálním síťovém segmentu
- **{pc_num}** – vrátí číslo počítače uložené v systémových proměnných

Zatímco formát se může v konečném produktu změnit, je důležité zachovat jejich funkcionalitu.

F4: Definice musí jít skládat do bloků, tyto bloky by dále měly umožnit nastavení závislosti na dalších blocích.

F5: Program umožní studentovi zobrazit definovanou otázku a odpovědět na ni.

F6: Dále pak umožní získat zpětnou vazbu a to buď:

- Ve formě automatické opravy.
- Nebo ve formě částečně automatické opravy.

F7: Program musí být schopen uložit stav odpovědí.

F8: Program musí být schopen obnovit uložený stav.

F9: Součástí implementace musí být síťové API, dále označováno jako monitorovací server. Tento monitorovací server musí být schopen shromažďovat údaje o zodpovězených otázkách.

F10: Monitorovací server musí být schopný zaslat tyto údaje ověřeným klientům.

F11: Kombinace monitorovací server/klient dále pak musí umožňovat manuální vyhodnocení těchto odpovědí.

3.2.3 Nefunkční požadavky

N1: Aplikace poběží pod distribucí linux - Ubuntu, verze 14.4 nebo vyšší.

N2: Aplikace by měla být použitelná i pod jinými distribucemi OS linux.

N3: Program pro testování musí být navržen tak, aby byl frontend a backend důkladně oddělen – náhrada frontendu by neměla být složitá.

N4: Součástí implementace by mělo být síťové API.

3.2.4 Případy užití

Přehled pokrytí jednotlivých funkčních požadavků případy užití lze nalézt v tabulce 1. Mapování aktérů k případům užití je vizualizováno na obrázku 2.

UC1: Definovat otázku – vyučující vytvoří adresář představující modul, následně uvnitř tohoto modulu vytvoří adresáře představující bloky. Uvnitř každého z bloků do textového souboru zapíše definici otázky nebo testu podle strukturované šablony.

UC2: Definovat správnou odpověď – vyučující v rámci definice do definičního souboru zapíše data validační šablony. Touto šablonou může být přesná odpověď

UC3: Manuálně vyhodnotit – v průběhu výuky laboratoří může student požádat vyučujícího o manuální ohodnocení některých typů úloh. Tedy úloh které nelze ohodnotit automaticky.

UC4: Načíst modul – na začátku výuky student otevře adresář s modulem aktuálního cvičení. Z tohoto adresáře si systém načte definici bloků, otázek a testů.

UC5: Načíst uložený postup – student nestihl úlohu dokončit na předchozím cvičení, proto si pomocí kontextové nabídky načte svůj uložený postup. Alternativně studenta potkalo selhání počítače, může si ale načíst alespoň svůj poslední uložený postup.

UC6: Odpovědět na otázku – student si vybere jeden z odemknutých bloků, a vybere otázku na kterou ještě správně nezodpověděl a chce na ni odpovědět. Svou odpověď potvrdí, systém tuto odpověď následně validuje proti definici.

UC7: Opravit špatnou odpověď – pokud student nezodpoví otázku správně, student může změnit svou odpověď a opakovat validaci dokud jeho odpověď neodpovídá definovanému validátoru.

UC8: Uložit postup úlohou – student si postup úlohou pravidelně ukládá interakcí s validací.

UC9: Zobrazit odpověď – student má po zodpovězení otázky přístup k odpovědi, která byla využita při její validaci.

UC10: Zobrazit definovanou otázku – student v klientské aplikaci nalezne výpis jednotlivých bloků modulu, některé z nich mohou být uzamčené. Každý blok uvnitř obsahuje informace o náplni bloku a samotné dříve definované otázky nebo testy. Každý z těchto dílků má svůj vlastní popis a nápovědu. Na každou otázku, která je v otevřeném bloku (a není úspěšně zodpovězena / v procesu kontroly) lze odpovědět, způsob odpovídání se může lišit v závislosti na typu pole, také lze spustit každý neprocházející a nespuštěný test.

UC11: Zobrazit stav vyhodnocení – student má přístup k stavu vyhodnocení poté co odešle požadavek na validaci. Vyhodnocen může být třemi stavy a to úspěch, neúspěch a čekání na vyhodnocení, kde stav čekání je přechodný. Vyučující stav vyhodnocení odpo-

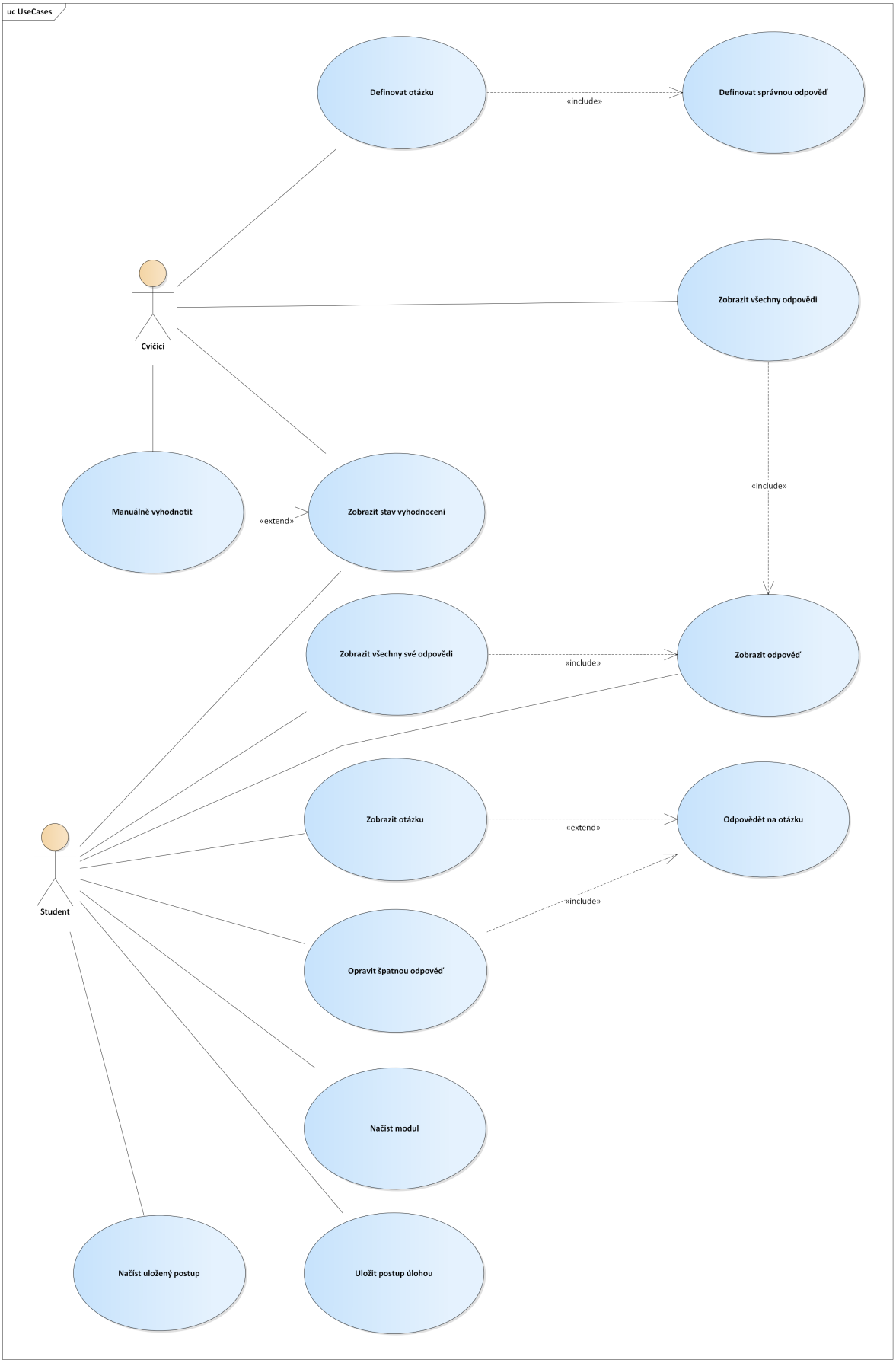
vědi studenta nalezne s pomocí svého klienta, v kolonce stav, na řádku odpovídajícího záznamu.

UC12: Zobrazit všechny odpovědi – vyučující se ve svém klientu provede načtení všech přítomných záznamů do jednoho seznamu. Další záznamy se v seznamu objeví dynamicky po posunutí zcela dolu a budou se objevovat dokud nejsou všechny záznamy vyčerpány.

UC13: Zobrazit všechny své odpovědi – student si své odpovědi může prohlédnout pod komponentou bloku. Jelikož nelze odpovídat na otázky v neodemčených blocích, bude mít k dispozici všechny své odpovědi.

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
UC1	X		X	X							
UC2			X								
UC3	X			X		X					X
UC4	X	X		X							
UC5	X	X		X				X		X	
UC6	X	X			X	X					X
UC7	X	X			X	X					X
UC8	X	X		X	X	X	X		X		
UC9	X	X			X						
UC10	X	X		X	X						
UC11			X	X		X		X		X	X
UC12								X		X	
UC13								X		X	

Tabulka 1: Pokrytí funkčních požadavků případy užití



Obrázek 2: Diagram případů užití

4 Praktická část

Kapitola popisuje zajímavé problémy, a otázky, které musely být vyřešeny, respektive zodpovězeny pro vytvoření výstupů této práce.

Nejprve je uvedena zvolená platforma, společně s programovacím jazykem. Ve zbývajících kapitolách jsou pak rozvedeny samostatné části projektu – studentský klient, monitorovací server a v poslední řadě monitorovací klient.

4.1 Výběr programovacího jazyku

Pro vývoj jsem zvolil programovací jazyk Scala, zvolil jsem jí díky její moderní syntaxi, silně objektovému přístupu a type inference. Oproti C++ se dá jednodušeji přenést, a délka vývoje v tomto jazyku je značně kratší. Díky tomu, že běží pod prostředím Java, je vysoce multiplatformní. Navíc v porovnání s .NET je Java minimálně na linuxových distribucích už léty ozkoušená a široce podporovaná. S nasazováním systémů pod Javou má navíc vedoucí práce zkušenosti, a tedy je mi schopen v případě problémů asistovat. Já mám rovněž s vývojem v Javě zkušenosti.

4.2 Studentský klient - Backend

Student je hlavním aktorem během laboratorních cvičení, pro zjednodušení práce je tento klient klíčový. Byť díky redukci zadání, má jeho realizace váhu podobnou zbytku realizovaných částí. Tento klient pomůže odesíláním odpovědí na serverovou část, kde bude možné otázky vyhodnotit manuálně, dále nám umožní formu automatického vyhodnocení otázek.

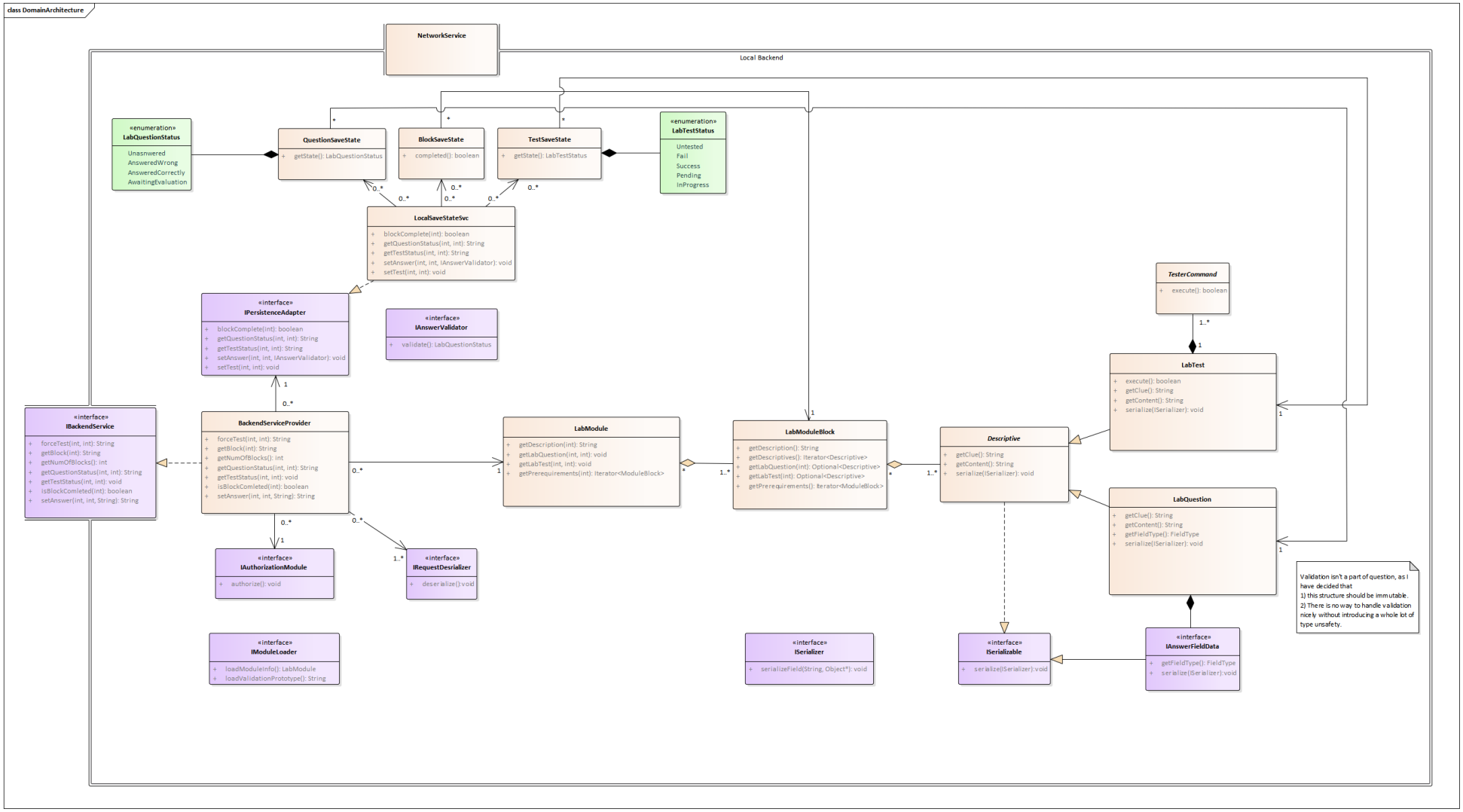
V této části je nejprve předestřen návrh aplikace, jako takové. V té souvislosti dále naleznete, jak bylo vyřešeno načítání datových struktur v neměnném (*immutable*) modelu.

4.2.1 Architektura a design

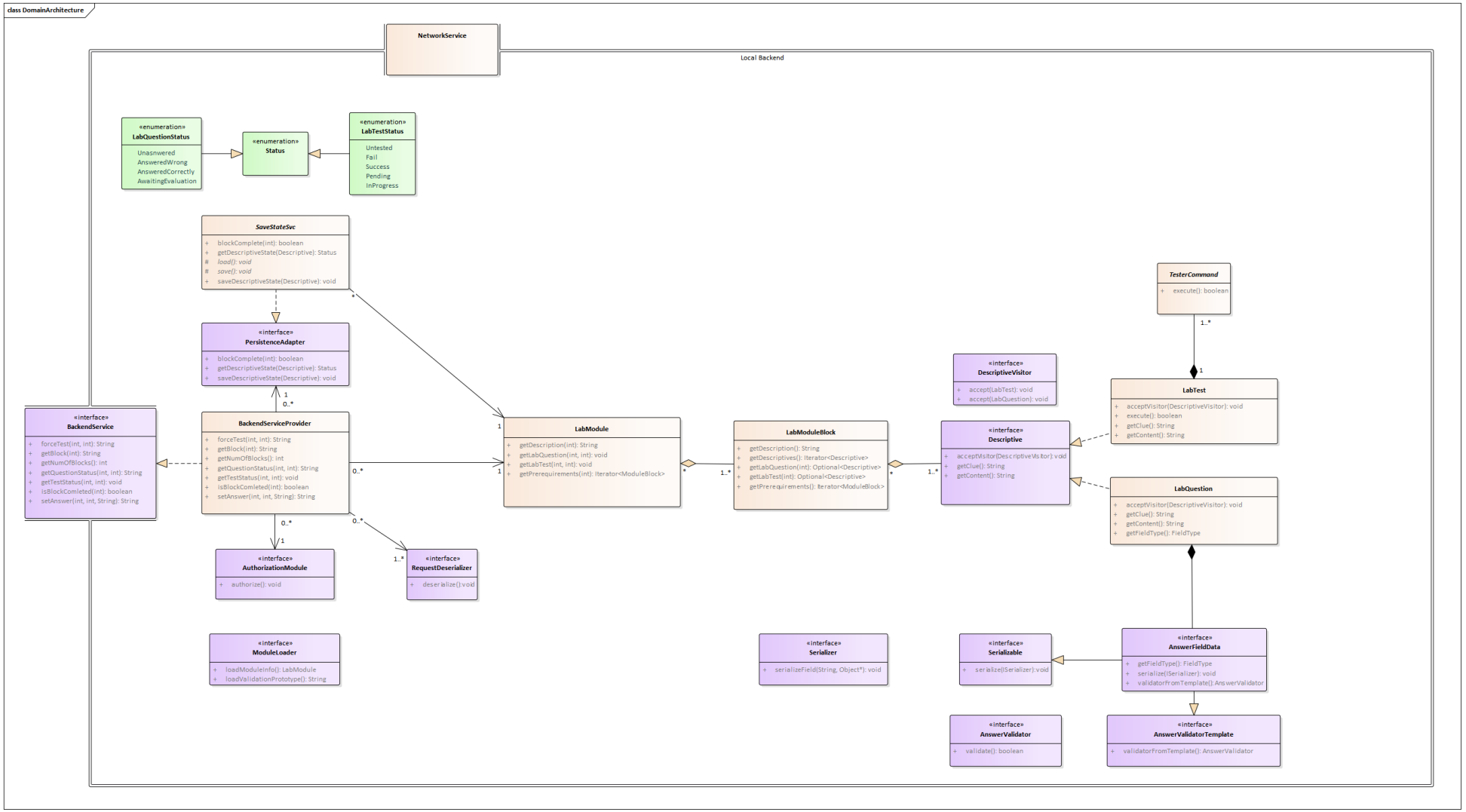
Iniciální návrh (viz. obrázek 3), se ukázal jako nevhodný. Největším problémem bylo modelování otázek a testů. Jak lze vidět, *LabModuleBlock* původně poskytoval otázky oddělené od testů, přestože měly oba prvky stejného předka. To bylo indikací, že s modely není něco v pořádku. Musím poznamenat, že už na začátku jsem zamýšlel pro tyto struktury neměnnost.

Jedním pádným důvodem pro určitou změnu byla nepřístupnost specifických vlastností testů, tento problém bylo možné vyřešit gettery vnitřní struktury, nicméně tím by zvýšila provázanost tříd. Dobrým řešením se zdál návrhový vzor *visitor*, díky Scale by se sice dal problém vyřešit i s pomocí pattern matchingu, ale zmíněný návrhový vzor se mi zdá jako čistější řešení.

Správné použití visitora vyžaduje, aby byl splněn předpoklad konečného počtu podtypů, respektive, je potřeba vědět, že se počet podtypů bude navyšovat buď zřídka nebo se nebude navyšovat vůbec. Tento předpoklad je v našem případě splněn, jelikož se v daných situacích budou měnit především kompozitní součástky *TesterCommand* nebo *AnswerFieldData*. Díky visitoru se můžeme zbavit rozšíření *Serializable* na rozhraní *Descriptive*, jelikož nyní už bude možné serializaci provádět externě.



Obrázek 3: První návrh klientské aplikace



Obrázek 4: Finální předimplementační návrh

4.2.1.1 Ukládání stavu

Jak je patrné z původního obrázku, návrh porušoval alespoň jeden z principů SOLID[2]. V případě *PersistanceAdapter*u konkrétně D – Dependency Inversion. Zprvu se nedařilo spojit neměnné struktury s validací odpovědí, jelikož jejich vazba byla velmi slabá. Zároveň se jednalo o případ, kdy její výstup závisí na vstupu uživatele. Při konstrukci objektu naneštěstí nemusíme mít tento vstup k dispozici, a proto se jako nejlepší postup zdálo předávat vyrobené validátory až po získání tohoto vstupu.

Eventuálně jsem se rozhodl pro přijetí abstrakce navzdory obtížnější konstrukci. Dle FTSE[3], neexistuje problém, který by nešel vyřešit další úrovní indirekce. Vytvořil jsem proto nové rozhraní: *AnswerValidatorTemplate*, které by mělo vytvořit validátor z předem získaných argumentů a nějakého vloženého vstupu. Tímto by měla být konstrukce abstraktně odložena, a tak je nyní možné využít jiných (už měnitelných) objektů, jenž posléze mohou vstup dosadit.

4.2.1.2 Backend service provider

Úplně prvně se celá idea návrhu točila okolo představy jednoho backendu na počítači, na něž by se připojil tenký klient. Toto rozhraní pak bylo hlavním bodem kontaktu mezi frontendem a backendem. Tento návrh (zobrazen na obrázku 4) kypí zbytečnou komplexitou, lepším řešením by byl démon, jenž by běžel v rámci frontendu, který by na pozadí prováděl potřebné operace validace či ověřování testů. Démon se ovšem ukázal také jako zbytečně komplexní. Finální návrh tedy využívá dalších vláken pouze při některých operacích, jakými jsou: validace, testování, ukládání stavu nebo sdílení výsledků.

Toto rozhraní tedy skončilo jako užitečná fasáda, nejprve bylo ale potřeba změnit výstupní typy, jimiž byl úplně v první iteraci řetězec. Důvodem pro to byla idea serializace, a to například do formátu JSON. Samotný frontend by pak pouze přijal a zpracoval danou informaci dle sebe. Jelikož takové řešení by zabralo čas navíc, za ne úplně velkých přínosů. Proto byla učiněna taková změna, díky niž nyní poskytuje přímo aplikační struktury. Kdyby v budoucnosti bylo potřeba změnit toto chování zpět na chování démona, nebo dokonce serverového backendu, lze napsat adaptér, který by využíval této fasády jakožto základu.

4.2.2 Načítání modelů

Během sběru požadavků vzešel návrh adresářové struktury modulu. Strukturu jsem implementoval podle požadavků.

Samotné definice bloků a jejich popisovatelných součástí jsem realizoval za pomoci souborů formátu YAML[4]. Jedná se o pro uživatele jednoduše čitelný formát, se kterým se dobře pracuje. Což je přesně to, co chceme u definic. Na rozdíl od své podmnožiny, formátu JSON, nevyžaduje při svém formátování závorky, které mohou snížit čitelnost.

Některé definice by šly zredukovat, například by šlo využít soubory v podsložce jakožto implicitní definice testů a otázek, nicméně potom není možné znovu využít soubory či referovat k souborům v jiných složkách. Toto je ovšem aspekt, jehož korektnost se ukáže až během využívání softwaru.

4.2.2.1 Využití abstraktních syntaktických stromů

Pro načítání struktury libovolného datového typu se hodí abstraktní syntaktické stromy. Aby měla aplikace co nejvyšší modularitu uvnitř vlastních modulů a nemusela spoléhat na struktury externích

2 Definice termínu SOLID: https://en.wikipedia.org/wiki/SOLID
3 Fundamental Theorem of Software Engineering
4 Specifikace formátu k dispozici z: https://yaml.org/

knihoven, bylo vhodné zavést strukturu, kterou by šlo zpracovávat nezávisle na použitém typu parseru. Každý parser vytváří abstraktní syntaktický strom, nicméně funkcionality vrcholů se mohou lišit. Pro tuto aplikaci postačí dvě operace – *getChild(key)* a *parse[T]*. Metoda *getChild(key)* nám získá libovolného potomka, a to na každém vrcholu, který není listem, tato metoda vyžaduje klíč, ten by měl být libovolného typu. Pokud by budoucí práce potřebovala vyměnit strukturu, do které se ukládají definice otázek a testů, lze číslovat například numericky, kdy pořadí může být libovolně zvolené pro každého potomka. *Parse[T]* je pak operace která umožní s pomocí úzkého setu pomocných datových struktur zpracovat aktuální podstrom jako konkrétní datový typ. Bez ohledu na to, kterou knihovnu pro parsování využijeme, by neměl být problém konkrétní strom překonvertovat do tohoto formátu.

4.2.2.2 Závislosti bloků

Menším požadavkem byla závislost bloků, tedy každý blok může definovat, které bloky musí být dokončeny pro své „odemknutí“. Při implementaci se ukázalo, že je problematické takové bloky zkonstruovat. Tento, jinak triviálně řešitelný, problém se vyskytl díky mému rozhodnutí o neměnitelnosti zmíněných bloků. Jelikož už při konstrukci by bloky měly přijmout definitivní, neměnitelný seznam těchto závislostí.

V klasických programovacích jazycích jako C či C++, by se tento problém dal řešit jednoduše, alokovat pole správné velikosti a následně využít adres uvnitř tohoto pole k vytvoření samotných objektů s pomocí ukazatelů. Toto řešení by šlo využít i ve vyšším jazyku C# s použitím klíčového slova *unsafe*. V prostředí Java ovšem tento přístup není možný. Jedním možným řešením je využití stylu funkcionálního programování, do konstruktoru podat seznam anonymních funkcí které vrací požadovanou hodnotu, alternativně funkci jenž vrací seznam instancí. Dalším možným řešením je tzv. „líná inicializace“. Potřebná instance je v takové situaci vytvořena až v momentě, kdy je poprvé hodnota či metoda volána, jakékoli další přístupy už vracejí stejnou hodnotu. V Javě je líná inicializace realizována s pomocí initialization-on-demand holder idiomu. Scala jakožto využitá technologie nabízí přímo zabudovanou realizaci líné inicializace při použití klíčového slova *lazy*.

Každé z předchozích řešení by uložilo seznam tak, jak byl definován. Dokážu si představit případy, za kterých by takový stav byl vyžadovaný. U této aplikace to ovšem tak není. Situace, kdy je blok závislý sám na sobě nebo obecně v cyklické závislosti, je nežádoucí. Proto využívám topologického řazení (topsortu) k postupnému vybudování bloků. Pokud se kruhová závislost v definicích vyskytne, modul ji vyhodnotí jako chybu a vyhodí výjimku.

Topsort lze realizovat pomocí několika algoritmů, já jsem zvolil variantu DFS. Zatímco jsem zvážil využití Kahnova algoritmu, data nejsou vhodně uzpůsobena (neexistuje žádný indikátor zdrojů). Data by sice šlo Kahnově algoritmu přizpůsobit, nicméně když máme k dispozici ekvivalentní způsob, pro který není třeba dělat takové změny, proč ho nevyužít.

4.2.2.3 Makra

U maker jsem narazil na koncepční problém, jejich rozšíření je možné pouze v případě, pokud jsou adresy nastaveny na správných rozhraních. Řekněme, že vyžadujeme makro které dosadí MAC adresu jiného PC, pak potřebujeme v první řadě připojení k síti, což je podmínka, která nemusí být splněna, proto může dosazení makra selhat.

Nutno připomenout, že makra mohou být dosazena i do některých polí otázek, proto nelze vyčkat až do momentu validace. Dosazování maker je možné odložit až do chvíle před zobrazením. Pakliže tak učiníme, je otázka, co dělat po zobrazení. Finální stav implementace tedy odloží dosazení co nejdéle to je možné. Pakliže se problém projeví, je vyvolána výjimka. Kompletní adresování problému je ale potřeba zařídit až v další práci.

4.3 Monitoring server

Cvičící by si měl vždy mít možnost prohlédnout postup studentů. Ve chvíli, když je systém plně automatický, může tak udělat jen tam, kde se uchovávají studentova data. Aplikace má ukládat studentův postup, nicméně tento požadavek je koncipován, aby se zabránilo ztrátě uživatelských dat, například při systémovém selhání. Jelikož je však potřeba sběr dat studentů, bude se ukládat stav na dva cílové body – lokálně a na server. Ze serveru se pak tyto údaje dají dále využívat nebo poskytovat (byť třeba anonymizovaně).

Server nám dále pomůže vyhodnotit otázky, které by se špatně převáděly do podoby textového pole nebo multichoice otázky. Server může otázky poskytnout cvičícím, kteří pak mohou otázky vyhodnotit manuálně. Server tedy pošle data, tak jako by je poslal kterémukoli jinému klientu.

4.3.1 Technologie

Jelikož tester je funkčně silně specializovaný, není na něj potřeba víc než několik knihoven. Pro server jsem se ale rozhodl využít framework. Hlavní důvodem je rychlá tvorba RESTového API. Řešení v podobě frameworků mají k dispozici všechny součásti pro přijímání a předzpracování požadavků klienta.

4.3.1.1 REST

Vzhledem k tomu jaké informace jsou na serveru ukládané, není potřeba složitějších API. GraphQL nebo Falcor, nejsou vhodné jelikož naše data nebudou strukturálně složitá, zatím tedy není třeba řešit problém N+1. Pro využití gRPC nebude mít aplikace, dle mého názoru, dostatečný počet mikroslužeb, aby se takový výběr dal odůvodnit.

Pro komunikaci s klienty taktéž není potřeba robustních protokolů jako je SOAP, naprosto si vystačíme s formátem JSON[5], přes protokol HTTP, respektive jeho zabezpečeným analogem HTTPS. REST nepředepisuje formát zpráv, není proto problém případně o formátu s klienty vyjednávat.

4.3.1.2 Scala, Play, databáze a ORM

Původně jsem měl v úmyslu využít Scalu a pro ni určený open-source framework Play. Rozhodnutí pro tento framework bylo relativně přímočaré. Pro tento jazyk je možností několik, nicméně s žádnou z nich jsem nebyl před zahájením práce obeznámen, takže jsem na základě několika seznamů vybral nejlépe vypadající možnost.

Vše vypadalo rozumně, do té chvíli než přišla chvíle na ORM. ORMs jsou široce využívané v enterprise architekturách, přesto jsou do určité míry kontroverzní, někdy i nazývané anti-patternem. Celá diskuze kolem této tematiky se točí kolem problému zvaného *object-relational impedance mismatch*. Dle Irelanda, Bowerse, Netwtona a Waugha je problémů několik: strukturální, instanční, zapouzdření, identity, modelu zpracování a vlastnictví. [8] Existujících implementace ORMs se navíc mohou potýkat s problémy výkonnosti nebo exekuce na více vláknech [9].

V reakci na tento problém ve Scale, jakožto jazyce se silnou komunitou funkcionálních programátorů, vznikla knihovna Slick. Tato knihovna si dala za cíl namísto toho, aby byla dalším ORM, se problému kompletně vyhnout. Vytvořila konceptuální přístup tzv. FRM (Functional relational mapping). V souvislosti se [8] FRM problémy ORM překonává přijetím funkcionálního paradigma [10], čímž všechny zmíněné problémy zmizí. Zároveň to ale znamená, že už nemanipulujeme s instancemi objektu, ale s *tuples*. To může a nemusí být problém, záviselo by na využití, pro tento projekt by to problém nebyl. Z praktického hlediska to znamená, že se s dotazem, který bude převe-

5 Specifikace k dispozici z: https://www.json.org/json-en.html

den na specifický dialekt SQL, dá pracovat jako se Scala kolekcemi řádků tabulky. Dále nepotřebujeme případně nadbytečné doménové entity. Tedy podobná myšlenka jOOQ[6] (pro čtenáře znající jazyk C# se jedná o Java ekvivalent LINQ[7]). Na první pohled se jedná o ideální stav – kolekce typově bezpěčných objektů, query-on-demand, třída tabulky jako čistokrevná šablona pro schéma (tedy žádné přebytečné modely).

Tento sen se ale rychle rozplynul. Pro to, aby bylo možné změnit používanou databázi bylo potřeba využít DAO. To není nic nezvyklého, spousta persistence řešení používá tento typ objektů. Nepříjemná část spočívala v propojenosti těchto objektů s definicí databázových tabulek. Tento typ objektů nešel generalizovat nebo jinak obecně shrnout pod jednotné rozhraní, jelikož definice tabulek musela být pevně součástí DAOs. Pro to, aby se dala tabulka definovat bez návaznosti na typ databáze, bylo potřeba znát profil Jdbc. Pro generalitu takového přístupu by bylo potřeba Jdbc profil používat i v částech aplikace, kam jednoznačně nepatřil. Pokud bych tak učinil, tak bych nedbal na patřičné oddělení zodpovědnosti (SoC).

4.3.1.3 Návrat k Javě, Spring, Spring Data JPA

Jelikož jsem se rozhodl jít cestou obyčejných ORM a Scala pro ně neměla patřičné prostředí, vydal jsem se jiným směrem. Už jsem několikrát dělal na projektu, jehož nedílnou součástí bylo RESTové rozhraní, jednou se shodou okolností jednalo o jazyk Java s frameworkem Spring[8]. Díky této zkušenosti jsem vybral stejné prostředí i pro tento subprojekt. Mimo jiné Spring poskytuje JPA[9], jako nadstavbu ORM sytému Hibernate.

Není to ovšem jediný důvod proč použít právě Spring, potažmo Spring Boot[10]. V první řadě Spring je široce rozšířený, díky tomu je poměrně jednoduché dohledávat informace týkající se jak API, tak častých problémů. Dále pak, jelikož má Spring živou komunitu, je pak jednodušší najít vývojáře na případný další vývoj. Pokud se i přesto sestavení zkušeného týmu nepovede, tým, který nemá s frameworkem žádné zkušenosti, se pak bude hladčeji aklimatizovat novému prostředí.

4.3.2 Architektura subprojektu

Při tvoření API jsem dodržoval principy vícevrstvé architektury – prezentační vrstvu zajišťují řadiče, business vrstvu pokrývají služby a persistenci pokrývají repozitáře společně s JPA. Kdybychom měli být důkladní tak navíc k tomu existují doménové/modelové entity s příslušnými DTO[11]s společně se JPA vrstvou. Dohromady pět vrstev s nízkou provázaností. Každá z vrstev komunikuje vždy nanejvýše se sousedními vrstvami.

Přiznávám, že bylo možné využít architekturu dvouvrstvou vynecháním servisní vrstvy. Takovou aplikaci by však bylo obtížné udržovat. Jakákoli úprava JPA dotazu by mohla způsobit vedlejší efekty v jednom a více řadičích. Dále by se obtížněji testovalo, neboť by bylo vyžadování rozsáhlejší mockování repozitářů. Posledními aspekty jsou duplicitní kód, případně bezpečnost. Duplicitní kód mluví sám za sebe, z hlediska bezpečnosti mám na mysli absenci DTOs (jejich použití, ale dvouvrstvá struktura nevylučuje). Kdyby aplikace odesílala přímo databázové objekty, dávala by tak hypotetickému útočníkovi cenné informace o struktuře databáze.

Monolitická aplikace by pak měla nejen nevýhody dvouvrstvé architektury, ale zároveň by bylo obtížné dělat obměnu databázového systému. Dále by porušilo podmínku RESTového API, tedy zra-

6 https://www.jooq.org/
7 https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/
8 https://spring.io/
9 https://spring.io/projects/spring-data-jpa
10 https://spring.io/projects/spring-boot
11 https://martinfowler.com/eaaCatalog/dataTransferObject.html

tilo by svou RESTovost. Monolitické aplikace se vyvíjejí jen v případech, kdy je vyžadován rychlý a levný vývoj řešení, kde není potřeba hledět na údržbu projektu.

4.3.3 Databázová struktura

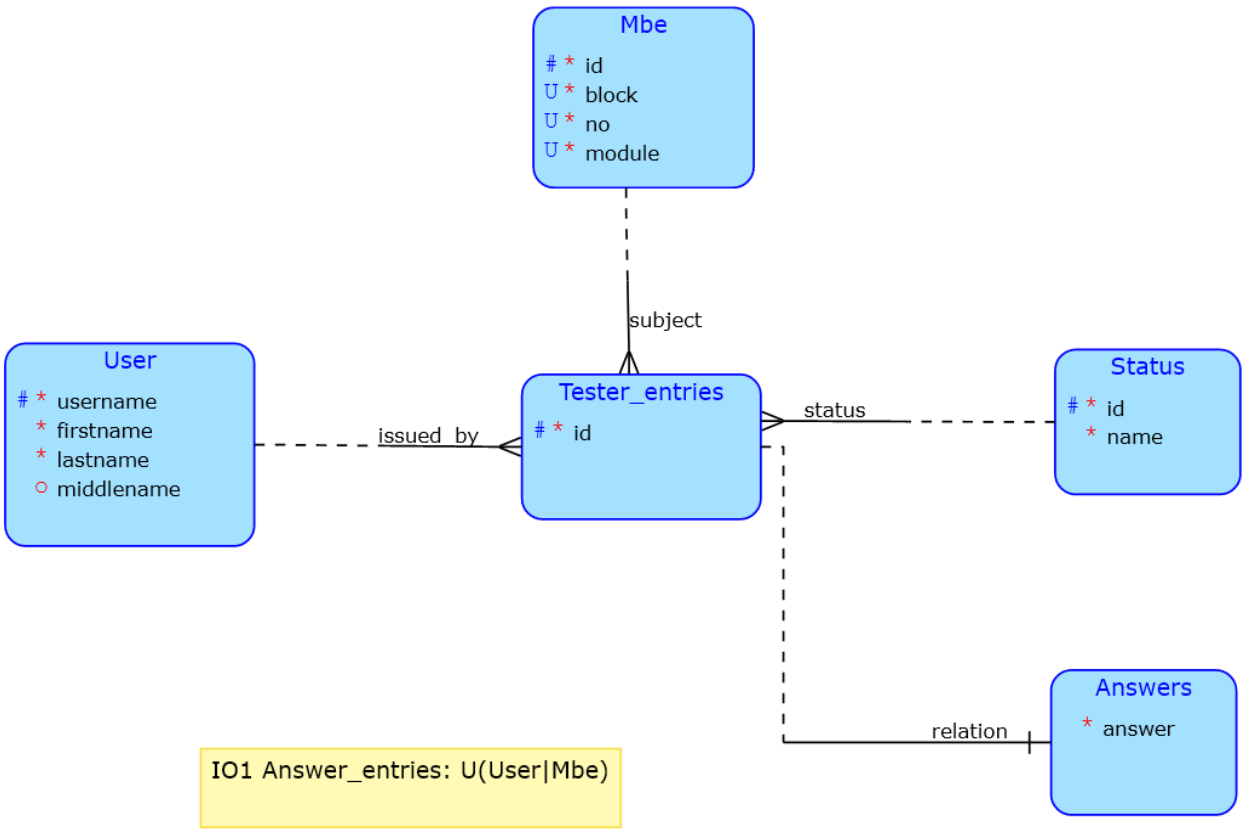
Databázovou strukturu tvoří 5 entit: uživatel, záznam, mbentita, status a odpověď (viz. Obrázek 5). Tato struktura vznikla z potřeb aplikace.

Uživatel je cache záznam uživatele, obsahuje pouze základní údaje, které nespadají pod ochranu osobních údajů. Tento uživatel by měl předem existovat v nějakém jiném (vzdáleném) repozitáři. Tento uživatel je uložen v lokální databázi, aby bylo možné získat záznam v případě výpadku vzdáleného repozitáře. Zároveň je pak o něco jednodušší realizovat některé testovací scénáře.

Záznam je entitou reprezentující záznam, který odeslalo testovací prostředí. Těmito informacemi jsou typicky: uživatel, kterého se záznam týká; modul, blok a číslo řadové popsané komponenty (*descriptive*); status, který určuje korektnost; časová známka a v některých případech odpověď.

Jak je patrné, předchozí entita byla dekomponována do několika dalších pomocných entit – konkrétně statusu, mbentity a odpovědi. Tato dekompozice bylo vytvořena, aby se předešlo duplikaci dat.

Přestože byly podniknuty kroky k odstranění duplikace dat, díky entitě odpověď, databáze není normalizovaná, a to ani podle první normální formy, neboť se jedná o strukturované pole. Pokud bychom, ale uznali tento případ jako pouhý text či blob, pak už by byla databáze normalizována podle BCNF.



Obrázek 5: Schéma databázových entit

4.3.4 REST API

Jelikož se jedná o RESTové API, je vhodné definovat zdroje se kterými budeme pracovat. Serverová aplikace potřebuje zaznamenávat údaje o provedených operacích. Těmito operacemi může být třeba spuštění testu nebo zodpovězení otázky.

Prvním zdrojem tedy bude kolekce *users*. Každý uživatel v kolekci by měl poskytnout základní informace, konkrétně by mělo stačit celé jméno. Jednotliví uživatelé by měli mít virtuální kolekci *entries*. V této kolekci jsou záznamy, které odeslal tento uživatel, v této kolekci lze listovat a vyhledávat dle data přijetí záznamu.

Uživatelé jako takoví, by neměli být přidáváni jakožto samostatná entita, ani jejich úprava by neměla být možná. Každý student, potažmo učitel se již nachází ve fakultní usermapě, není tedy přípustné, aby se dělaly pouze lokální změny. Uživatele by sice mělo jít přidat, nicméně by tak mělo jít učinit buď skrze proces vytváření záznamu (včetně patřičných ověření identity) nebo s pomocí jiných interních procesů.

Druhým zdrojem je kolekce všech záznamů *entries*. Každá odpověď o záznamu obsahuje informace jako: kdo záznam uložil, součástí kterého modulu, bloku a testu či otázky (tato trojice je označována jako MBN) se záznam týká, dále pak jeho stav, případně konkrétní odpověď. V této kolekci jde vyhledávat pomocí data vytvoření, jednotlivých atributů MBN a uživatelského jména, dále je v ní možné listovat.

Třetím zdrojem je podmnožina záznamů, která k sobě má patřičnou odpověď – *answers*. Operacemi a strukturou je totožná s minulým zdrojem.

V poslední řadě, aby se předešlo duplikaci dat, jsou záznamové entity uloženy odděleně a každá má svoje vlastní identifikační číslo. Zatímco věřím, že většina situací tento koncový bod nebude vyžadovat, určitě se může hodit mít k dispozici překlad z MBN na toto číslo a naopak. Tento překlad bude k dispozici pod kolekcí *aggregates*, jakékoli další operace, či vybírání podmnožin neshledávám jako nezbytné.

4.3.4.1 Řadiče záznamů a odpovědí

Spring Boot poskytuje velmi příhodné anotace pro vytváření řadičů *@Controller* a speciální *@RestController* pro REST architektury. Řadič v rámci Springu figuruje jako komponenta s dodatečnými anotacemi. Každá takto definovaná komponenta je pak zároveň automaticky po konstrukci mapovaná, obsahuje-li kořenovou anotaci *@RequestMapping* nebo její potomky. Je-li toto označení použito na úrovni třídy, bude cesta definovaná v rámci anotace prefixem všem anotovaným metodám. Stane-li se však, že je nějaká třída takto označena a automaticky zkonstruována, jakákoli další konstrukce selže kvůli duplicitnímu mapování.

V API jsem definoval koncový bod pro podmnožinu odpovědí, tento bod by měl být vyjma výsledku, všemi operacemi identický k bodu *entries*. Jediným rozdílem z venčí by tedy bylo pouze mapovaní URL. Díky automatickému mapování při konstrukci ovšem nebylo možné novou instanci řadiče vytvořit.

Řešením tedy bylo odebrat jak anotace pro řadič, tak kořenovou anotaci *@RequestMapping*. Nyní ovšem přišla výzva jak metody řadiče namapovat. Spring samozřejmě poskytuje dva „out-of-the-box“ způsoby jak zajistit mapování, nicméně jeden slouží pouze pro servlety, nikoli pro řadiče, a druhý mapuje jenom metody. Můj cíl byl replikovat proces mapování bez nutnosti dělat tuto činnost manuálně nebo tvoření subtypu rozhraní.

Jelikož Spring automaticky provádí mapování k anotacím, bylo jasné, že má třídu, kterou pro tento účel využívá. Tou třídou je *RequestMappingHandlerMapping.* Tato třída sice nemá rozhraní, které

by tento proces zprostředkovalo, ale má užitečnou metodu *#getHandlerMappings(handler)*, třída zároveň dává přístup k výsledkům mapování. Z vnějšku je k těmto mapováním možné dodat prefix řadiče. Pro realizaci je tedy nejprve potřeba zpřístupnit tuto protected metodu do lokálního kontextu, což bylo realizováno rozšířením třídy.

4.3.4.2 Identity

Každá databázová tabulka koresponduje s doménovou entitou. Domlouvat se ale mezi službami za pomocí primárních klíčů tabulek není optimální. Takový návrh má dva vážné nedostatky: v první řadě je náchylný ke změnám – pokud se změní například typ klíče, je potřeba měnit rozhraní služeb; v druhé řadě není možné kontrolovat, zda byl předán správný typ argumentu. Pro lepší ilustraci si lze představit dvě entity A a B s klíči A_k, respektive B_k typu *t*, pokud budu mít službu co je určena pro A_k, ale pracuje s klíčem typu *t*, nic mi nebrání jako argument vložit B_k. Použitím speciálních identitních typů pro každou relevantní databázovou entitu se vyhneme oběma zmíněným nedostatkům.

4.3.4.3 Poskytovatel identit – od výjimek k optional

Jelikož identita není primitivní datový typ bylo potřeba vytvořit tvůrce těchto identit. Nejjednodušším řešením je tvorba identit uvnitř řadičů. Tato tvorba ovšem velmi rychle narazí na problém s duplicitním kódem (porušení principu DRY).

Řešením je poskytovatel identit, jeho výhodou není jenom redukce duplikace, ale dovoluje delegovat zodpovědnost za jejich tvorbu. Některé struktury lze vytvořit, když nejsou nalezeny, jiné nikoli. Krásou pak je, že je možné mít rozdílné poskytovatele s různým chováním, třeba pro různé skupiny uživatelů.

Otázkou tedy zůstává, co dělat když není identita nalezena. Možností je několik: vrátit *null*, vrátit speciální *NotFound* identitu, vyhodit výjimku nebo prohlásit hodnotu za optional. Vracení null hodnoty není vhodné, jelikož je potřeba proti ní kontrolovat, částečně tím porušuje kontrakt metody čímž přidává nutnost dokumentace. NotFound identita je lepší řešení v tom, že neporušuje kontrakt, požaduje však modifikaci identitního rozhraní, aby se dala kontrolovat validita identity. Výjimka je rozumným řešením, zachováváme-li princip fail-fast. Za nešikovnou vlastnost tohoto řešení se dá označit narušení aplikačního toku. Nejlepším řešením dle mého názoru je optional, hned z předpisu je vidět, kterou hodnotu volající získá, i že ji získat nemusí, čímž se metoda sebe-dokumentuje.

4.3.5 Zabezpečení, autentizace a autorizace

Jelikož se student nachází v laboratoři počítačových sítí, je třeba počítat s tím, že každá stanice má na sobě nainstalované potřebné diagnostické nástroje. Mezi těmi je i nástroj Wireshark, který se využívá pro zaznamenávání síťové komunikace. Díky této skutečnosti je potřeba komunikaci šifrovat, aby se při odchycení komunikace nedostaly uživatelská data do špatných rukou.

Pro zajištění šifrování využívá server protokolu HTTPS, systému X.509 s RSA klíči. Implementace poskytuje vlastní certifikační autoritu pro účely vývoje a testování. Produkční server by tento certifikát používat neměl.

V učebně mohou vyskytnout zvídaví a vynalézaví studenti. Mohlo by se proto stát, že se pokusí na základě informací z konfiguračních souborů prozkoumat RESTové koncové body. Tím by mohli získat přístup k datům svých spolužáků. Nejedná se sice o data utajovaná ani citlivá, nicméně se jedná o data soukromého charakteru, tedy nemělo by dojít k porušení důvěrnosti. Únik dat je menší problém v porovnání s narušením jejich integrity. Bez autorizace a autentizace by měl kdokoli možnost změnit záznam jakéhokoli studenta, to je nepřípustné.

Pro autentizaci uživatele řešení využívá fakultního OAuth2 serveru[12], konkrétně se provádí introspekce tokenů poskytnutých klientem. OAuth2 je navržen jako autorizační protokol, za normálních okolností by tedy bylo možné z něj získat informace o uživateli, nicméně fakultní server poskytuje pouze mizivé informace o rolích uživatele. Pro získání potřebných autorizačních údajů se server ptá usermap API[13].

4.3.6 Serializace

Serializaci do nejběžnějšího přenosového formátu pro REST architekturu – JSON zajišťuje ve Spring Boot knihovna Jackson. Tato knihovna je sama o sobě dobře konfigurovatelná, s mnohými přizpůsobeními serializačního i de-serializačního procesu.

Se serializací úzce souvisí DTOs, jelikož jde o objekty které putují mezi uživatelem a serverem. Ty jsem ve většině řadičích dosazoval jakožto rozhraní. Díky tomu pak můžeme případně přidat konkrétní implementaci i pro jiné formáty než je JSON. Samozřejmě by bylo možné mít implementaci jen jednu, nicméně tu by bylo snadné „znečistit“ mnohými anotacemi. Toto znečištění je dalším důvodem, proč je dobré mít pro tyto účely rozhraní, jsou naprosto izolovány od serializační/de-serializační logiky a jejich anotací.

V prostředí Spring není možné Jackson knihovnu použít izolovaně, a pokud ano, pak jsem nebyl schopen nenalézt způsob jak. Standardní postup pro využití této knihovny spočívá v získání speciálního *Jackson2ObjectMapperBuilder*u. Tento builder je pak většinou použit jako výchozí *ObjectMapper*. Vytvoření izolované instance *ObjectMapper*u ignoruje jakékoli nastavení. Toto podivné chování může velmi ztížit testování rozhraní, jelikož takový *ObjectMapper* může vytvářet zprávy které nejsou validní, ku příkladu *Optional* hodnoty může odeslat jako komplexní datový typ, jenž vyjadřuje, že je hodnota nepřítomná (namísto její vynechání).

Kde je toto chování ještě zvláštnější je při de-serializaci zpráv, kdy Spring využívá *MappingJackson2HttpMessageConverter,* přestože by se dalo očekávat že bude pro tento proces využit primární *ObjectMapper* (tedy ten vytvořený výše popsanou metodou, případně ten označený *@Primary*), není tomu tak. Implementace využívá svůj vlastní. Kde ovšem začíná bizarní chování, je právě v případě využití už někdy zbudovaného mapperu. Takový mapper totiž z nějakého důvodu, v této specifické třídě, nemá ve své vnitřní cache správně přiřazený serializér Optional. Namísto toho využívá obyčejného JSON serializéru, čímž mapuje naprosto totožně s chybně nakonfigurovaným mapperem, konfigurace na tento stav nemá vliv. Úprava nastavení proto musí být učiněna až po vytvoření instance mapperu. Jakákoli úprava před vytvořením instance způsobí její následnou nefunkčnost.

4.3.7 Centrální řešení výjimek

U serverových aplikací může být složitý aplikační tok. Obecně hlídat každou servisní výjimku je zbytečný kód v řadičích. Navíc málokdy chceme hlídat vyhazování výjimek, pokud se vyskytne problém se kterým si aplikace neporadí je třeba selhat jak nejrychleji to jde. V případě serverové aplikace ale klient potřebuje dostat zpětnou vazbu, není patřičné aby dostal zpět HTTP kód 500. Tento kód signalizuje, že server na něco nebyl připraven, ale takové nevalidní oprávnění je něco na co server připraven je, dokonce musí.

Pokud používám pro tuto kontrolu jednotné rozhraní, pro kritické selhání jiná možnost než výjimka není. Změna rozhraní na víc expresivní typy je sice možnost, nicméně to je řešení jen tehdy, kdy se z výjimky lze zotavit.

12 Dokumentace k dispozici z: https://rozvoj.fit.cvut.cz/Main/oauth2
13 Dokumentace k dispozici z: https://rozvoj.fit.cvut.cz/Main/usermap-api

Pakliže se z výjimky nelze zotavit, je pořád potřeba dodat uživateli důvod selhání, málokdy má takové selhání původ na serveru. Spring zajišťuje odchytávání výjimek, každá aplikace si může zaregistrovat vlastní odchyt využitím anotace *@ControllerAdvice*, neposkytne-li aplikace tento odchyt jakákoli neodchycená výjimka je automaticky namapována na kód 500.

Tato aplikace odchytává některé výjimky služeb, způsobené třeba nedostatečnými oprávněními, ale i způsobené konflikty při vkládání záznamů.

4.4 Autorizační modul

Specifikace OAuth 2.0 označuje naše API jako *resource server*. Pro komunikaci s naším resource serverem se potřebují klienti autentizovat. Nejbezpečnějším tokem dat v OAuth 2 je tzv. *authorization code flow* (a jeho rozšířená verze PKCE). Tento tok byl použit, pro autorizaci klientů. Jeho bezpečnější rozšíření jsem nevyužil, neboť dokumentace fakultního API[12], a ani manažer API[14] nezmiňuje, že by byl tento tok podporován. Nicméně uznávám, že bylo možné se na tuto podporu zeptat osobně, nebo emailem, naneštěstí implementace byla rozsáhlá, a proto jsem neměl čas tak učinit.

Pro implementaci jsem využil několika knihoven – sttp client pro idiomatického HTTP klienta, http4s pro interní HTTP server (Scala vlastní server neposkytuje a v Javě je k dispozici pouze server z balíku sun, který není oficiálně podporován [40]) a sttp-oauth pro zjednodušení práce s implementací.

4.4.1 Bezpečnostní omezení

Jelikož implementace nevyužívá PKCE, existuje bezpečnostní riziko. Toto riziko nemá vysokou pravděpodobnost proměny v hrozbu, nicméně je přítomno. Jelikož lokální server, nutný pro získání autorizačního kódu, běží jenom pod protokolem HTTP, pokud dojde k odchycení lokální komunikace, tak může dotyčná osoba získat autorizační token. Tuto komunikaci je ale možné odchytit pouze na místním počítači, jelikož se pro zisk tokenu využívá přesměrování.

Jediná situace, kdy je tento stav nebezpečný je v případě, že je na počítači přítomna škodlivá aplikace, která by odchytávala komunikaci. Pokud však token získá osoba pro kterou byl token vydán, nejedná se o problém.

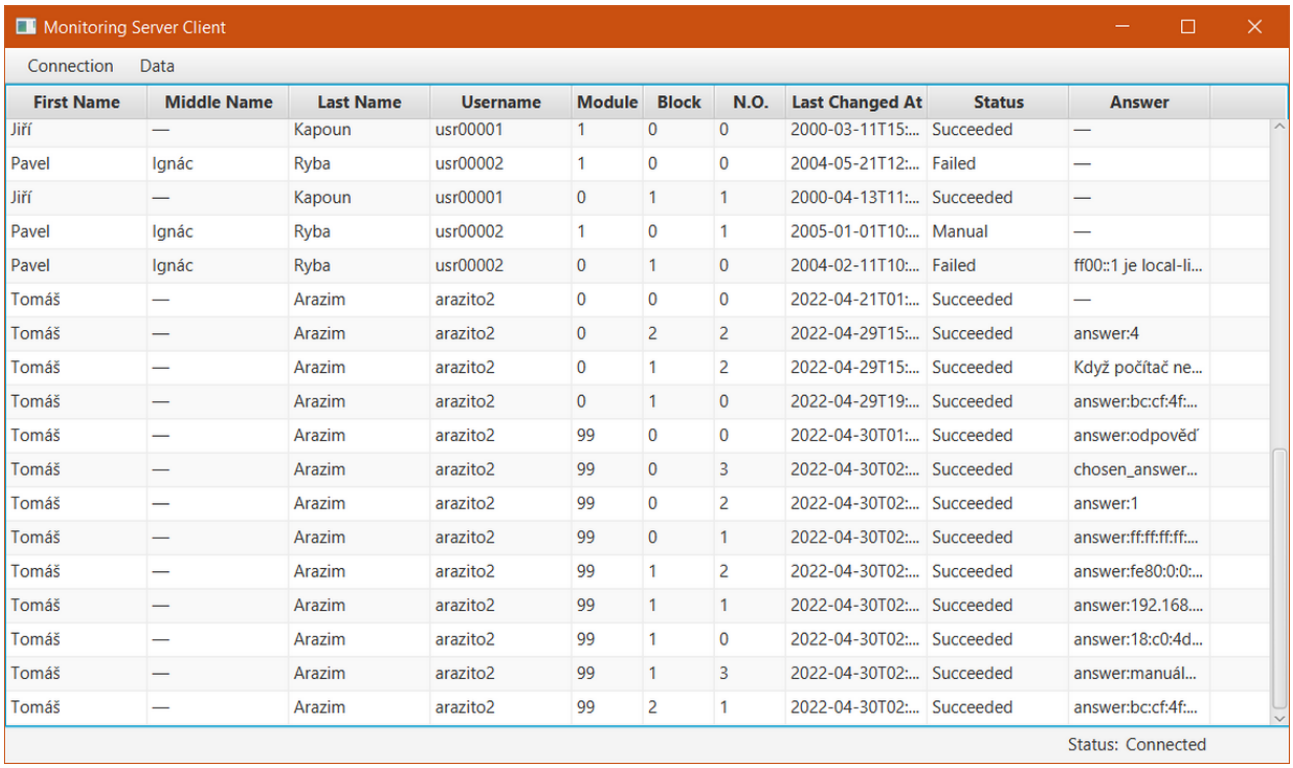
14 https://auth.fit.cvut.cz/manager/index.xhtml

4.5 Monitorovací klient

Server shromažďuje údaje o studentském postupu, tedy aby bylo možné prozkoumat jak se studentům daří při postupu modulem, je potřeba mít klienta, který si některé tyto informace vyžádá. Pro tento účel nám poslouží monitorovací klient.

Použití tohoto klienta je podmíněno rolí v rámci usermap ČVUT. Jelikož to, které konkrétní role by měly mít přístup, závisí čistě na nastavení serverové aplikace, může tohoto klienta teoreticky použít kdokoli, nicméně přístup k chráněným zdrojům mu může být zamítnut.

Díky návrhu API se jedná o velmi jednoduchou a relativně rychlou záležitost, především jelikož se bude jednat pouze o tenkého klienta. UI bylo řešeno s pomocí knihovny ScalaFX[15], což je idiomatický obal kolem knihovny JavaFX[16]. Aby se navíc snížil čas vývoje, využil jsem knihovnu ScalaFXML[17], která zjednodušuje tvorbu řadičů pro UI a výsledný kód činí více idiomatickým.



First Name	Middle Name	Last Name	Username	Module	Block	N.O.	Last Changed At	Status	Answer
Jiří	—	Kapoun	usr00001	1	0	0	2000-03-11T15:...	Succeeded	—
Pavel	Ignác	Ryba	usr00002	1	0	0	2004-05-21T12:...	Failed	—
Jiří	—	Kapoun	usr00001	0	1	1	2000-04-13T11:...	Succeeded	—
Pavel	Ignác	Ryba	usr00002	1	0	1	2005-01-01T10:...	Manual	—
Pavel	Ignác	Ryba	usr00002	0	1	0	2004-02-11T10:...	Failed	ff00::1 je local-li...
Tomáš	—	Arazim	arazito2	0	0	0	2022-04-21T01:...	Succeeded	—
Tomáš	—	Arazim	arazito2	0	2	2	2022-04-29T15:...	Succeeded	answer:4
Tomáš	—	Arazim	arazito2	0	1	2	2022-04-29T15:...	Succeeded	Když počítač ne...
Tomáš	—	Arazim	arazito2	0	1	0	2022-04-29T19:...	Succeeded	answer:bc:cf:4f:...
Tomáš	—	Arazim	arazito2	99	0	0	2022-04-30T01:...	Succeeded	answer:odpověď
Tomáš	—	Arazim	arazito2	99	0	3	2022-04-30T02:...	Succeeded	chosen_answer...
Tomáš	—	Arazim	arazito2	99	0	2	2022-04-30T02:...	Succeeded	answer:1
Tomáš	—	Arazim	arazito2	99	0	1	2022-04-30T02:...	Succeeded	answer:ff:ff:ff:ff:...
Tomáš	—	Arazim	arazito2	99	1	2	2022-04-30T02:...	Succeeded	answer:fe80:0:0:...
Tomáš	—	Arazim	arazito2	99	1	1	2022-04-30T02:...	Succeeded	answer:192.168....
Tomáš	—	Arazim	arazito2	99	1	0	2022-04-30T02:...	Succeeded	answer:18:c0:4d...
Tomáš	—	Arazim	arazito2	99	1	3	2022-04-30T02:...	Succeeded	answer:manuál...
Tomáš	—	Arazim	arazito2	99	2	1	2022-04-30T02:...	Succeeded	answer:bc:cf:4f:...

Obrázek 6: Uživatelské rozhraní monitorovacího klienta

Ve výsledném UI (na obrázku 6) je možné dvojklikem na sloupec status změnit výsledek daného záznamu. O každém záznamu jsou zobrazeny podstatné informace, jako kdo jej odeslal, k jakému modulu záznam náleží, kdy byl záznam naposledy změněn a i samotná odpověď u relevantních záznamů.

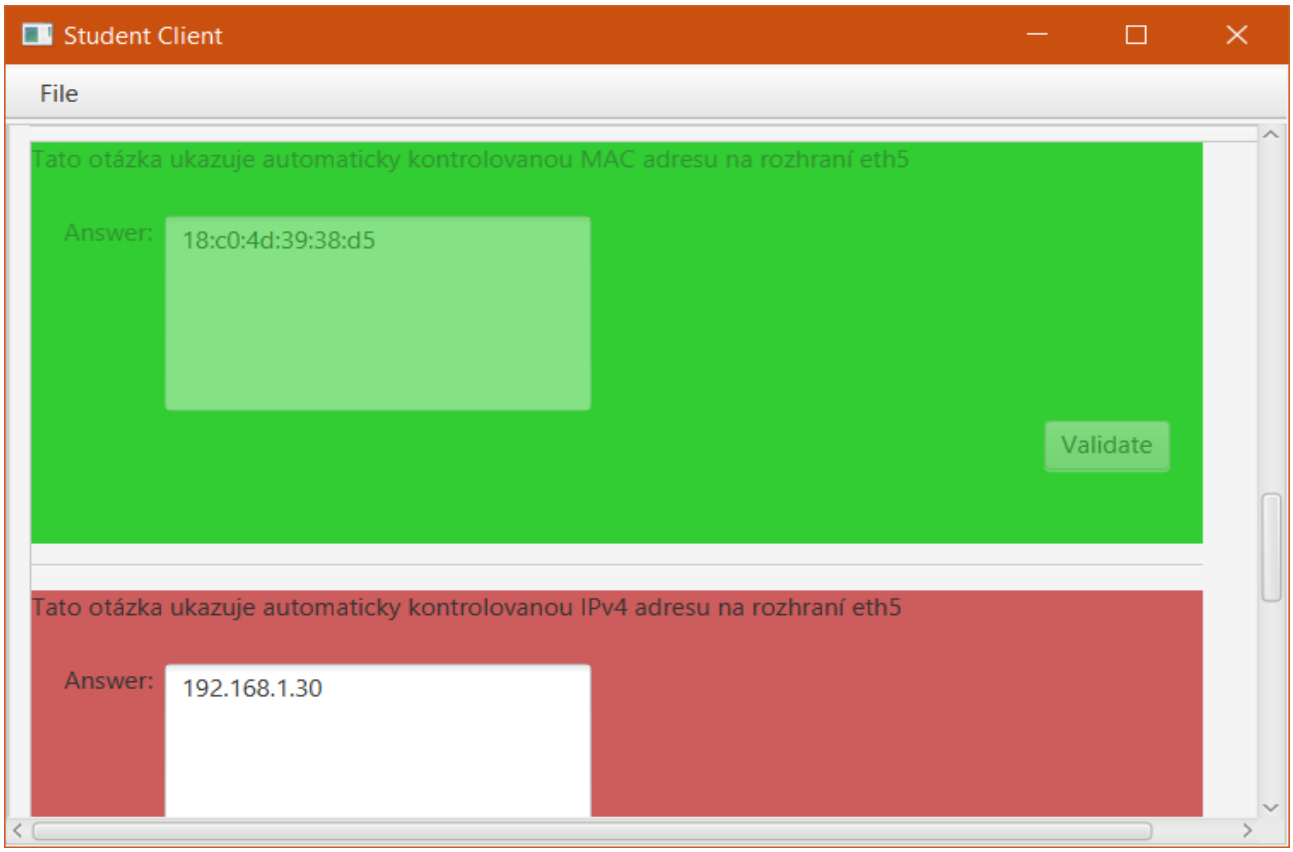
15 Domovská stránka projektu: https://www.scalafx.org/
16 Domovská stránka projektu: https://openjfx.io/
17 Projekt na GitHubu: https://github.com/vigoo/scalafxml

4.6 Studentský klient – Frontend

Stejně jako pro monitorovací klient jsem využil knihovnu ScalaFX (idiomatický obal kolem knihovny JavaFX), dále knihovnu ScalaFXML.

Zamykání modulů je koncept, který bylo potřeba zavést na úrovni frontendu, jinak by mohly být do kódu zavedeny chyby způsobené stavem. Například kdyby ukládací modul uložil stav, jenom pokud by byl odemčen, pak si troufnu tvrdit, že neplní pouze jednu funkci.

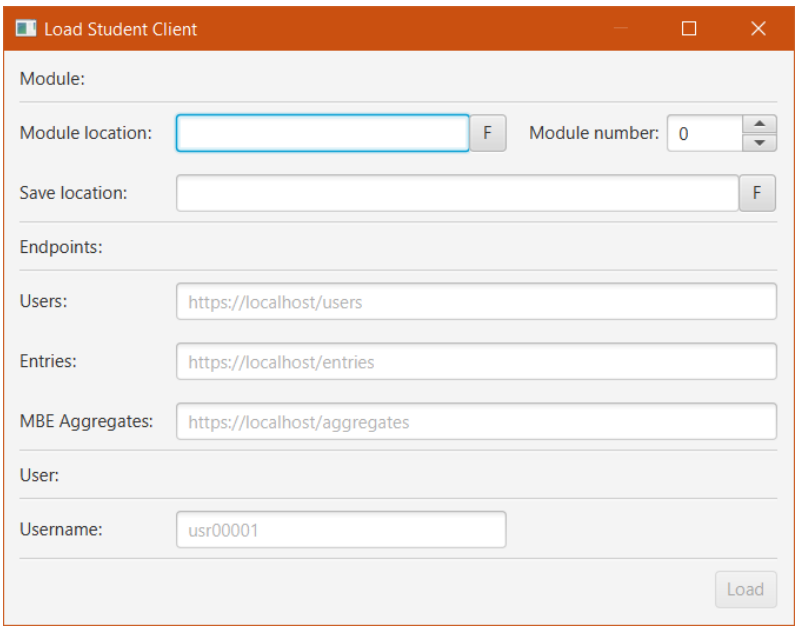
Pokud ovšem toto chování bude v budoucnu potřebné, není problém poskytnou další adaptér na backendové implementaci.



Obrázek 7: Uživatelské rozhraní studentského klienta

Uživatelské rozhraní (na obrázku 7) zprostředkuje načtení definovaných bloků, poté z nich vygeneruje několik rozbalovacích prvků. V tuto chvíli jsou podporované otázky typu textové pole a zaškrtávací tlačítko. Dále je připravený vizuál pro test, nicméně ten zatím nic nedělá. Po správném zodpovězení všech otázek v bloku jsou odemčeny všechny bloky, které tímto mají splněny všechny předchůdce.

Pro zprostředkování modulu je potřeba vyplnit lokaci definice modulu a zvolit kam se má uložit stav. Zároveň je potřeba navolit číslo modulu, pod tímto číslem se odesílá modul na server. Dále je potřeba nastavit koncové body serverové aplikace. Přestože je potřeba tyto body zadat, není nutné, aby byl klient během používání připojen k síti, lokální testování bude nadále k dispozici. (pro představu viz. obr. 8)



Obrázek 8: Dialog načítání modulu

4.7 Testování

Testování Scala projektů bylo realizováno s pomocí knihoven ScalaTest[18], ScalaCheck[19] a Mockito[20]. Většina kódu, podstatného pro produkci, je pokryta jednotkovými testy. Díky knihovně ScalaCheck jsou některé testy automaticky generované, svou cenu ukázaly například při ověřování ukládání stavu, kdy odkryly chybu při ukládání stavu bloků. Mezi všemi testy se vyskytuje i několik testů integračních.

Situace, pro které by se hodilo mít více integračních testů by vyžadovaly mít k dispozici serverovou implementaci. Takové nasazení by ale muselo být ochuzeno o autentizaci. Testy s autentizací na serveru mohou probíhat díky Spring Boot knihovně, která s modulem zabezpečení spolupracuje, nepokouší se ho vyřadit. Pokud by se měly testy odehrávat s autentizací, vyžadovalo by to autorizační token, ten aby fungoval automaticky, by vyžadoval dodatečný autorizační proud – *client credentials*. Pakliže bychom chtěli použít *authorization code flow*, bylo by potřeba mít testovacího uživatele, za kterého se lze přihlásit (toho bohužel nemám). Navíc autorizační server není součástí aplikace, ani jej nevlastním, a proto není vhodné proti němu dělat automatické testy. Pro absenci testů systémových je důvod totožný.

Jedním z východisek pro zajištění těchto typů testů by bylo nasazení vlastního autorizačního serveru, tam se ale dostáváme do teritoria za hranicí práce. Automatizované systémové testy (a více integračních) by sice zvýšily pokrytí a snížily šanci na výskyt chyb, nicméně při nasazení vlastního autorizačního serveru existuje riziko nekompatibility se serverem fakultním a jejich implementace by přidala, odvážím se vyjádřit odhad, měsíc implementace navíc. Pro otestování funkcionality jsem si proto vystačil s manuálním systémovým testem.

Pokrytí testy implementace backendu testeru je ~70% pokrytí řádek kódu, důvodem pro toto nízké pokrytí je přítomnost datových nebo implicitních tříd, jejichž implementace jsou triviální a netřeba je testovat. Velká „neotestovaná“ část balíku *service* je objekt, který se používá na frontendu, a jeho automatickým otestováním bychom zajistili velkou část systémového testu. Posledními neotestovanými částmi jsou companion objekty některých tříd nebo třídy, které ve finále nebyly využity.

18 Domovská stránka projektu: https://www.scalatest.org/
19 Domovská stránka projektu: https://scalacheck.org/
20 Domovská stránka projektu: https://site.mockito.org/

Server je pokryt řadou jednotkových a integračních testů. I tak přiznávám, že by se hodilo více integrace v oblasti databáze. Pro jejich realizaci byly využity knihovny JUnit 5[21], spring-security-test[22] a spring-boot-starter-test[23]. Testy systémové byly opět prováděny manuálně. Pokrytí zde je ~80%. U repositářů si lze všimnout, že je pokrytí 100% (0/0), je to ale jen díky tomu, že Spring Boot dovoluje využití rozhraní pro definici repozitářů a rozhraní se do pokrytí nezapočítávají. Dále si lze povšimnout, že balík *security* není tak důkladně otestovaný. Tam se vyskytl kód, který je určen pouze pro účely vývoje – konkrétně kód pro obdržení tokenu v případě, že není k dispozici jiný klient. Jelikož jsou tyto třídy (třídy *CVUTZuulOAuth2UserService* a *OAuth2SuccessTokenResponseHandler*) jen pro účely vývoje, nebylo je potřeba testovat.

Balíček	Tříd	Metod	Řádků
core	50% (6/12)	68% (24/35)	68% (26/38)
exception	76% (10/13)	71% (10/14)	73% (11/15)
fields	57% (4/7)	53% (7/13)	60% (9/15)
loader	60% (39/64)	72% (168/233)	73% (337/461)
persistance	58% (35/60)	71% (160/225)	73% (343/469)
serializer	66% (2/3)	19% (5/26)	17% (5/29)
service	25% (1/4)	42% (9/21)	17% (13/73)
utility	100% (2/2)	100% (5/5)	85% (6/7)
validator	63% (12/19)	83% (51/61)	85% (69/81)

Tabulka 2: Pokrytí testy backendu studenského klienta

Balíček	Tříd	Metod	Řádků
configuration	100% (7/7)	90% (20/22)	96% (56/61)
controller	80% (4/5)	97% (37/38)	97% (119/122)
data	75% (12/16)	78% (70/89)	79% (139/175)
exception	100% (6/6)	60% (6/10)	55% (10/18)
repository	100% (0/0)	100% (0/0)	100% (0/0)
security	80% (8/10)	65% (25/38)	54% (124/228)
service	100% (15/15)	98% (84/85)	95% (271/283)

Tabulka 3: Pokrytí testy serverového API

End-to-end testy byly taktéž prováděny manuálně, jelikož jejich automatická realizace by vyžadovala všechny součásti implementace, včetně autorizace. Automatizace těchto testů by byla přínosná, nicméně by vyžadovala značnou časovou dotaci (součástí by byly i systémové testy).

21 Domovská stránka projektu: https://junit.org/junit5/
22 K dispozici z: https://mvnrepository.com/artifact/org.springframework.security/spring-security-test
23 K dispozici z: https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-test

5 Výsledky

Z implementační části vznikly následující výsledky:

1) Serverová RESTful aplikace, která shromažďuje záznamy o výsledcích testů. API umožňuje jejich úpravu oprávněným uživatelem a tím jejich manuální vyhodnocení. Dále je vystavena sada dalších přístupových bodů, díky níž mají budoucí rozšíření v podobě klientů možnost požádat o potřebné informace. Komunikace je připravená běžet pod protokolem HTTPS. Server je zabezpečen autentizací oproti fakultnímu OAuth2 serveru, autorizace přihlášených uživatelů je zajištěna oproti API ČVUT usermap. Použitý framework zajišťuje, společně s realizovanou částí, jednoduchou rozšiřitelnost API.

2) Tenký klient, který zprostředkovává základní komunikaci pro cvičící laboratoří se serverovou aplikací. Tento klient si žádá data z testů a umožňuje změnu statusu (hodnocení). Taktéž zprostředkovává jednoduché filtrování. Pro přístup na server využívá OAuth2 tokenu, který získává z fakultního serveru.

3) Tlustý klient, který studentovi ukáže předem definované, do bloků uspořádané otázky a umožní na ně odpovědět. Tyto otázky mohou obsahovat makra, která doplní specifické proměnné závislé na prostředí. Stav odpovědí se ukládá jak na monitorovací server, tak lokálně. Tento klient je připraven na budoucí rozšíření ve směru testování studentských konfigurací, ale samotné testování není jeho součástí. Frontend a backend tohoto klientu je pečlivě oddělen, takže je možné vyměnit uživatelské rozhraní.

6 Diskuze

Díky širokému rozsahu požadavků je velký prostor pro rozšíření. Nejužitečnějším by zajisté bylo testovací prostředí, které by cvičícím ušetřilo další kus práce. Už nyní se ale jedná o přínos, který by měl snížit vytížení vyučujících v průběhu laboratorních cvičení. Dále nyní existuje API, které je možné rozšířit a vytvořit s jeho pomocí interaktivnější výuku.

Další práce by mohla adresovat některé designové problémy, na které se přišlo až během vývoje. V první řade, co dělat v případech, kdy se počítač není schopen připojit k segmentu, ze kterého chce vyžádat informace o MAC adrese. Pokud by se takové makro ukázalo v otázce, co s ní a čím ji nahradit. Zároveň lze uvažovat i rozšíření tohoto systému na jednoduchý jazyk, který by dovolil používat některé výrazy, jako je sčítání nebo odčítání uvnitř definic, přistupovat k některým proměnným atd.

Druhá věc, kterou by bylo dobré adresovat je ukládání celých odpovědí. Nejedná se o největší problém, jelikož pokud odpověď byla ohodnocena jako korektní, pak ji není třeba měnit. Pakliže byla ohodnocena jako chybná, je beztak potřeba odpověď opravit. Jednalo by se tedy pouze o *quality-of-life* vylepšení.

Dále pak lze uvažovat o vylepšení UX; o modulech distribuce, administrace aktivních modulů nebo gamifikaci / sociálních prvcích. Mezi méně potřebné změny by se dalo uvažovat o websocket serveru, díky čemuž by se nemusel na klientech využívat polling při čekání na ohodnocení.

Vylepšení marginální by znamenalo například přidat některé integrační a systémové scénáře, specificky na serverové aplikaci přímo nad stejným typem databázového systému, jako bude použit na produkci. Dále pak optimalizace některých databázových operací při tvorbě záznamu.

U některých tříd monitorovacího klienta by se taktéž dal udělat refaktoring, který by pomohl v přehlednosti, dále by bylo vhodné dopsat některé chybějící jednotkové testy.

7 Závěr

Cílem práce byla analýza laboratorních úloh, na základě které měla být vypracována část systému pro jejich automatickou kontrolu. Konkrétně se jednalo o serverovou aplikaci společně se dvěma klientskými aplikacemi rozdílné tloušťky.

Nejprve byla provedena analýza úloh síťových laboratoří. Výsledky analýzy byly posléze prokonzultovány s vedoucím práce, na základě této konverzace byla sestavena řada funkčních a nefunkčních požadavků. Cíl analýzy laboratorních úloh byl tedy splněn.

Jelikož analýza nalezla významný počet požadavků, byl z tohoto seznamu vybrán jeho podmnožina. Z této podmnožiny jsem navrhl v několika iteracích design studentského klienta. Následně byla vytvořena kostra, a implementace některých funkcí. Před implementací komunikačních součástí byla realizována serverová část.

Přestože původně nebyla v plánu komplexní serverová aplikace, v zájmu modularity, rozšiřitelnosti a udržovatelnosti bylo namísto toho vytvořeno normalizované REST API s vícevrstvou architekturou. Finální aplikace je navíc zabezpečená, a kontroluje autoritu uživatelů. Díky těmto skutečnostem se ovšem délka implementace protáhla. Dokončením tohoto bodu je splněna druhá část cíle o serverové aplikaci běžící na učitelském počítači.

Jelikož monitorovací klient byl pouze tenký klient, nebylo třeba na něj alokovat mnoho času. Během vývoje navíc pomohl odhalit některé, hlavně konfigurační problémy na serverové části. Bohužel byl tvořen ve zrychleném tempu, proto některé jeho části nemohou býti staveny k standardu zbytku projektu. Na druhou stranu, jelikož se jedná o tenkého klienta, jeho náhrada je formalitou.

Ve finále byl dokončen tlustý klient. Díky tomu, že jsem měl připravenou velkou část implementace, jeho realizace se zdařila včas, čímž byla splněna první část druhého cíle.

Splněnost třetího cíle nemohou objektivně posoudit, jelikož dle mého názoru jsem principy softwarového návrhu dodržoval. Mnoho času bylo věnováno návrhu, dle něhož je aplikace dobře testovatelná, rozšiřitelná a udržovatelná. Troufnu si tedy tvrdit, že třetí cíl byl splněn.

Poslední cíl bohužel splněn nebyl. K splnění tohoto bodu nemohlo dojít neboť implementace byla dokončena až ke konci semestru a laboratoře síťových cvičení byly, a v době psaní stále jsou ke konci semestru přetížené. Tedy nemohlo dojít na odsimulování úloh na cvičení, ani na otestování v ostrém provozu. Jedná se tedy v tuto chvíli pouze o demo, nicméně vedoucí práce se domnívá, že testování bude moci bez velkých problémů proběhnout během zimního semestru.

Seznam odborné literatury

[1] HLINĚNÝ, Petr. *Základy teorie grafů* [online]. 1. vyd. Brno: Masarykova univerzita, 2010, s. 1-2. [cit. 2022-03-30]. Dostupné z: http://is.muni.cz/elportal/?id=878389. ISSN 1802-128X.

[2] HLINĚNÝ, Petr. *Základy teorie grafů* [online]. 1. vyd. Brno: Masarykova univerzita, 2010, s. 14. [cit. 2022-03-30]. Dostupné z: http://is.muni.cz/elportal/?id=878389. ISSN 1802-128X.

[3] KUNDEL, Dominik. ASTs - What are they and how to use them. In: *twilio.com* [online]. 11. 6. 2020. [cit. 2022-03-28]. Dostupné z: https://www.twilio.com/blog/abstract-syntax-trees.

[4] HLINĚNÝ, Petr. *Základy teorie grafů* [online]. 1. vyd. Brno: Masarykova univerzita, 2010, s. 8. [cit. 2022-03-30]. Dostupné z: http://is.muni.cz/elportal/?id=878389. ISSN 1802-128X.

[5] KOVÁŘ, Petr. Definice sítě. KOVÁŘ, Petr. *Úvod do Teorie grafů* [online]. Vysoká škola báňská – Technická univerzita Ostrava a Západočeská univerzita v Plzni, ©2021, s. 127-131. [cit. 2022-05-07]. Cesta k vědění (Academia). Dostupné z: https://homel.vsb.cz/~kov16/files/uvod_do_teorie_grafu.pdf

[6] KAHN, A. B. Topological sorting of large networks. *Communications of the ACM* [online]. 1962, **5**(11), 558-562. [cit. 2022-04-22]. ISSN 0001-0782. Dostupné z: doi:10.1145/368996.369025

[7] CORMEN, Thomas H. Section 22.4: Topological sort. *Introduction to algorithms*. 2nd ed. Boston: McGraw-Hill, ©2001, s. 549-552. ISBN 0-262-03293-7.

[8] IRELAND, Christopher, David BOWERS, Michael NEWTON a Kevin WAUGH. A Classification of Object-Relational Impedance Mismatch. *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications* [online]. IEEE, 2009, 2009, 36-43. [cit. 2022-03-19]. ISBN 978-1-4244-3467-1. Dostupné z: doi:10.1109/DBKDA.2009.11

[9] RUSSELL, Craig. Bridging the Object-Relational Divide. *Queue* [online]. 2008, **6**(3), 18-28. [cit. 2022-03-18]. ISSN 1542-7730. Dostupné z: doi:10.1145/1394127.1394139

[10] Introduction. *Scala-slick.org* [online]. Typesafe, 2015. [cit. 2022-03-25]. Dostupné z: https://scala-slick.org/doc/3.0.0/introduction.html

[11] Java Downloads for All Operating Systems. *Java.com* [online]. Oracle. [cit. 2022-04-30]. Dostupné z: https://www.java.com/en/download/manual.jsp

[12] Java™ Platform Overview. *Oracle Help Center* [online]. Oracle, ©1993-2022. [cit. 2022-05-01]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/index.html

[13] What is .NET?. *.NET | Free. Cross-platform. Open Source.* [online]. Microsoft, ©2022. [cit. 2022-05-01]. Dostupné z: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet

[14] Download .NET 6.0. *.NET | Free. Cross-platform. Open Source.* [online]. Microsoft, ©2022. [cit. 2022-05-01]. Dostupné z: https://dotnet.microsoft.com/en-us/download/dotnet/6.0

[15] HUNTER, Scott. .NET Core is the Future of .NET. In: *.NET Blog* [online]. Microsoft, ©2022, 2019. [cit. 2022-05-01]. Dostupné z: https://devblogs.microsoft.com/dotnet/net-core-is-the-future-of-net/

[16] BAMBURIC, Mihăiță. .NET Framework is dead -- long live .NET 5. In: *Betanews* [online]. BetaNews, ©1998-2022, 2019. [cit. 2022-05-01]. Dostupné z: https://betanews.com/2019/05/07/future-of-dotnet/

[17] LANDER, Rich. LICENSE. In: *Github* [online]. .NET Foundation, ©2022, 2017. [cit. 2022-05-01]. Dostupné z: https://github.com/dotnet/core/blob/main/LICENSE.TXT

[18] ISO/IEC 23271:2012. *Information technology — Common Language Infrastructure (CLI)*. 3. International Organization for Standardization, 2012.

[19] ECMA-335. *Common Language Infrastructure (CLI)*. 6. Ecma International, 2012.

[20] SETIA, Varun. Understanding .NET Framework, .NET Core, .NET Standard And Future .NET. In: *C# Corner: Community of Software and Data Developers* [online]. C# Corner, ©2022, Mar 11, 2020. [cit. 2022-05-01]. Dostupné z: https://www.c-sharpcorner.-com/blogs/understanding-net-framework-net-core-and-net-standard-and-future-net

[21] RAMEL, David. Microsoft Survey: Developers Held Back by Lack of 'Native AOT' in .NET Core. In: *Visual Studio Magazine* [online]. 1105 Media, ©1996-2022, 31. 8. 2020. [cit. 2022-05-01]. Dostupné z: https://visualstudiomagazine.com/articles/2020/08/31/aot-survey.aspx

[22] .NET TEAM. CoreCLR is now Open Source. In: *.NET Blog* [online]. Microsoft, ©2022, 2015. [cit. 2022-05-01]. Dostupné z: https://devblogs.microsoft.com/dotnet/coreclr-is-now-open-source/

[23] CARTER, Phillip. [.NET Core]: .NET Goes Cross-Platform with .NET Core. *MSDN magazine* [online]. Microsoft, 2016, **31**(4). [cit. 2022-05-01]. ISSN 1528-4859. Dostupné z: https://docs.microsoft.com/en-us/archive/msdn-magazine/2016/april/net-core-net-goes-cross-platform-with-net-core

[24] Write once, run anywhere?. In: *ComputerWeekly.com* [online]. TechTarget, ©2000-2022, 02 May 2002. [cit. 2022-05-04]. Dostupné z: https://www.computerweekly.com/feature/Write-once-run-anywhere

[25] The Java Language Environment. *Oracle.com* [online]. Oracle, ©2022. [cit. 2022-05-04]. Dostupné z: https://www.oracle.com/java/technologies/introduction-to-java.html

[26] Java (programming language). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, ©2001-2022 [cit. 2022-05-04]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=1085290127

[27] Tour of Scala: Introduction. *Scala-lang.org* [online]. Lausanne (EPFL) Lausanne, Switzerland: École Polytechnique Fédérale, ©2002-2022. [cit. 2022-05-04]. Dostupné z: https://docs.scala-lang.org/tour/tour-of-scala.html

[28] A tour of C#: Overview. *Microsoft Docs* [online]. Microsoft, ©2022, 18. 3. 2022. [cit. 2022-05-05]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

[29] STROUSTRUP, Bjarne. *The C++ programming language*. 3rd ed. Reading: Addison-Wesley, 1997. x, 910 s. ISBN 0-201-88954-4.

[30] STROUSTRUP, Bjarne. The Essence of C++ [přednáška]. In: *Youtube* [online]. George Square Lecture Theatre, Edinburgh: The University of Edinburgh, 28 April 2014. [cit. 2022-05-05]. Záznam dostupný z: https://www.youtube.com/watch?v=86xWVb4XIyE.

[31] STROUSTRUP, Bjarne. C++ Applications. *Stroustrup.com* [online]. Bjarne Stroustrup. [cit. 2022-05-05]. Dostupné z: https://www.stroustrup.com/applications.html

[32] ISO/IEC 14882:2020. *Programming languages — C++*. 6. International Organization for Standardization, 2020.

[33] GUDGIN, Martin, Marc HADLEY, Noah MENDELSOHN, Jean-Jacques MOREAU, Henrik F. NIELSEN, Anish KARMARKAR a Yves LAFON, ed. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition): W3C Recommendation. *World Wide Web Consortium (W3C)* [online]. W3C, 2007, 27 April 2007. [cit. 2022-05-05]. Dostupné z: https://www.w3.org/TR/soap12-part1/#intro

[34] FIELDING, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, 2000. Dissertation. University of California.

[35] NELSON, Bruce J. *Remote Procedure Call*. Palo Alto, California, 1981. Dissertation. Carnegie-Mellon University.

[36] Introduction to gRPC: Overview. g*RPC* [online]. gRPC Authors, ©2022. [cit. 2022-05-06]. Dostupné z: https://grpc.io/docs/what-is-grpc/introduction/

[37] landing-2 [obrázek]. In: g*RPC* [online]. gRPC Authors, ©2022. [cit. 2022-05-06]. Dostupné z: https://grpc.io/img/landing-2.svg

[38] What is Falcor?. *Falcor* [online]. Netflix, ©2015-2021. [cit. 2022-05-06]. Dostupné z: https://netflix.github.io/falcor/starter/what-is-falcor.html

[39] Introduction to GraphQL. *GraphQL* [online]. The GraphQL Foundation, ©2022. [cit. 2022-05-06]. Dostupné z: https://graphql.org/learn/

[40] FAQ - Sun Packages. *Oracle* [online]. Oracle, ©2022. [cit. 2022-05-06]. Dostupné z: https://www.oracle.com/java/technologies/faq-sun-packages.html

[41] GUPTA, Lokesh. REST API – N+1 Problem. *REST API Tutorial* [online]. Lokesh Gupta, © 2022, September 30, 2021. [cit. 2022-05-08]. Dostupné z: https://restfulapi.net/rest-api-n-1-problem/

Příloha A

Příručka nasazení

Jelikož se implementace skládá ze tří částí, bude nejprve popsáno nasazení serverové aplikace a poté nasazení klientů. Aby bylo možné aplikace nasadit je nejprve potřeba je zkompilovat.

Nasazení serverové aplikace

Pro nasazení aplikace lze využít buď přímé spuštění pomocí nástroje Gradle nebo spouštěcí soubor JAR. V případě, že by bylo nutné vytvořit soubor typu WAR, by bylo potřeba přizpůsobit konfiguraci Gradle.

Sestavení serveru

Požadavky:

- Java Development Kit verze 11

Postup:

1) Navigujte příkazovou řádku do podsložky ./monitoring-server/ kořenového adresáře projektu.
2) V závislosti na vašem operačním systému zadejte příkaz:
   - [Windows] `./gradlew.bat build`
   - [Linux] `./gradlew build`
3) Pokud si nepřejete spustit aplikaci přímo přes nástroj gradle wrapper, zadejte:
   - [Windows] `./gradlew.bat bootJar`
   - [Linux] `./gradlew bootJar`

Krok 3 vytvoří JAR soubor v podadresáři ./monitoring-server/build/libs/.

Spuštění aplikace

Požadavky:

- Java Runtime Environment verze 11.
- Databázový systém MariaDB (tento systém lze změnit, ale to vyžaduje zásah do sestavení)

Postup:

1) Navigujte se do podsložky ./monitoring-server/ kořenového adresáře projektu.
2) Nalezněte soubor *application.yml*, upravte tuto konfiguraci tak, aby souhlasila s nastavením vašeho databázového serveru. Během tohoto kroku lze změnit i další nastavení, jako jsou práva nebo lokace certifikátů.
3) Pokud zamýšlíte spouštět server přímo z gradle wrapperu, zadejte příkaz:

- [Windows] `./gradlew.bat bootRun`
- [Linux] `./gradlew bootRun`

4) Pokud chcete spustit server z JAR souboru zadejte `java -jar lokace_jar_souboru`.

Nasazení klientů

Distribuci je možné realizovat formou zvanou *fat JAR*. Jedná se o JAR soubor, který obsahuje všechny potřebné závislosti. Všechny použité knihovny používají Apache licence 2.0, která dovoluje distribuci se zahrnutou licencí a *copyright notice*. Tyto formáty ale nejsou součástí řešení, především také proto, že klienty je potřeba zkompilovat na koncovém operačním systému, kvůli knihovně JavaFX, jejíž moduly jsou závislé na prostředí.

Kompilace klientů

Požadavky:

- Nástroj SBT[24] verze 1.3.13.
- Java Development Kit verze 11.

Postup:

1) V příkazové řádce se navigujte do kořenového adresáře projektu.
2) Zadejte příkaz `sbt compile`.
3) Následně příkaz `sbt assembly`.

Tyto příkazy vygenerovaly soubory fat JAR v každém podprojektu. Spustitelné soubory obou klientů lze najít ve složkách ./tester-frontend/target/scala-2.13/ respektive ./monitoring-client/target/scala-2.13/.

Spouštění zkompilovaného klienta

Požadavky:

- Zkompilovaný klient.
- Java Runtime Environment verze 11.

Postup:

1) V příkazové řádce se navigujte do složky se zkompilovaným klientem.
2) Zadejte příkaz `java -jar lokace_jar_souboru`.

24 Postup instalace lze nalézt na: https://www.scala-sbt.org/1.x/docs/Setup.html

Uživatelská příručka

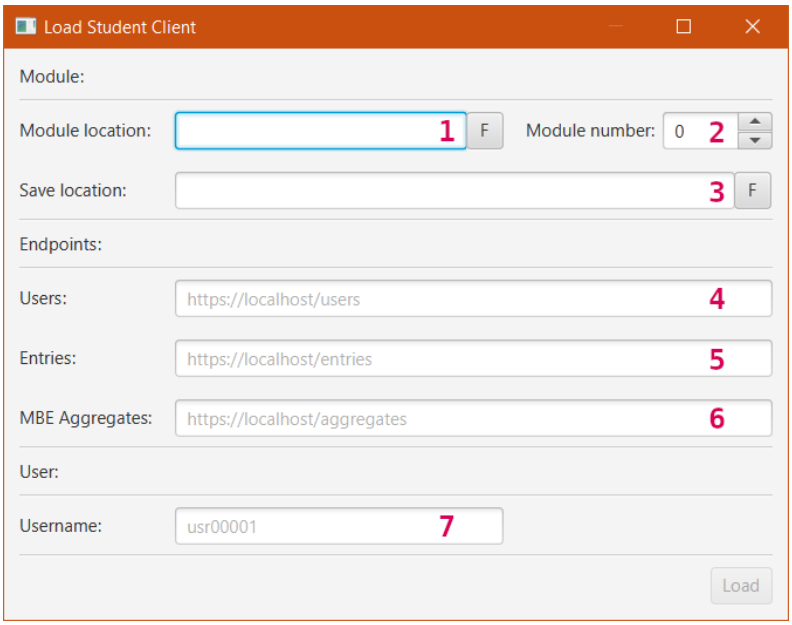
V této příručce se lze dočíst o ovládání klientských aplikací. Dále je uvedena i krátká kapitola o konfiguraci serveru.

Studentský klient

Tento klient je určen především pro použití studenty. Aby se dal klient použít, je potřeba mít k dispozici definici modulu. Jak vytvořit modul je popsáno v kapitole **vytváření modulů**. Pokud zatím nemáte žádný svůj modul k dispozici, je možné využít ukázkové moduly z adresáře ./tester-frontend/src/test/resources/.

Načítání modulu

1) Spusťte aplikaci.
2) V horní nabídce vyberte File > Open Module.



3) Na této obrazovce je potřeba zadat údaje o modulu, který chcete načíst.
   - Do kolonky označené 1 lze zadat lokaci adresáře modulu na vašem počítači. Alternativně lze stisknout tlačítko označené F, které otevře dialog výběru adresáře. Tato lokace musí existovat.
   - V kolonce číslo 2 navolte číslo modulu, tento krok je důležitý v případě že chcete svoje výsledky odeslat na serverovou část aplikace. Pokud zadáte špatné číslo modulu, uložené údaje mohou být v konfliktu s těmi novými. Detekce této situace je omezená, proto se ujistěte, že zadáváte správné číslo.
   - Do kolonky 3 zadejte jméno souboru do kterého se uloží stav vašich odpovědí. Alternativně lze stisknout tlačítko označené F, které otevře dialog uložení souboru.

- Do kolonek 4, 5 a 6 zadejte názvy koncových bodů pro uživatele, záznamy a agregátní typ RESTové části aplikace. Musí se jednat o validní webové URL, není ale nutné, aby byl počítač schopen se na tyto adresy připojit.
- Do kolonky 7 zadejte své uživatelské jméno, pod tímto jménem je potřeba se přihlásit. Pokud se přihlásíte pod jiným uživatelem, ukládání stavu na server selže autorizační chybou.

4) Klikněte na tlačítko Load.
5) V prohlížeči se vám po několika vteřinách zobrazí stránka autorizačního serveru https://auth.fit.cvut.cz/, zde vyplňte své přihlašovací údaje a klikněte na tlačítko přihlásit.
6) Pokud váš modul neobsahuje žádné chyby, uvidíte v okně aplikace definované bloky, otázky a testy. Alternativně se vám zobrazí chybová hláška.

Vytváření modulu

Modul se definuje za pomocí bloků. Pro každý blok by měl být vyčleněn adresář libovolného jména. Aby byl adresář rozpoznán coby blok, musí v sobě mít soubor se jménem `block_definition.yaml`. V tomto souboru lze definovat popis, nutné předpoklady a všechny otázky a testy, které budou součástí bloků. Modul s alespoň jedním validním neprázdným blokem je validní (zároveň ale musí být každý neprázdný blok validní).

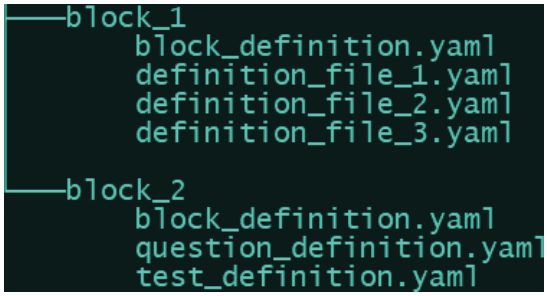
Podrobný postup:

1) Vytvořte adresář, na jeho jménu nezáleží.
2) Posuňte se do vytvořeného adresáře.
3) Vytvořte pro každý blok vlastní adresář, na jejich jménech nezáleží.
4) V každém z adresářů vytvořte soubor `block_definition.yaml`. Tento soubor má následující formát:

```
description: V tomto testovacím bloku si ukážeme vlastnosti klienta,
které mohou být závislé na vnějším prostředí. Otázky tohoto bloku jsou
z části automaticky kontrolované poslední je manuální.
descriptives:
  - ./definition_file_1.yaml
  - ./definition_file_2.yaml
  - ./definition_file_3.yaml
prerequisites:
  - block_1
```

   Pole *prerequisites* je nepovinné. Dodržte formát souboru YAML, vezměte v potaz, že příklad využívá zalamování řádků.

5) Váš modul může vypadat takto:



6) Obsah souboru definujícího otázku může vypadat takto:

```
content: Tato otázka ukazuje automaticky kontrolovanou IPv4 adresu na
rozhraní eth5
clue: Tato fráze se objeví, ve chvíli, kdy se jedná o špatnou odpověď
field type: Text
validator: "{host_ip:[eth4]}"
```

Každý validátor podporuje speciální hodnotu *{{MANUAL}}*, která označuje, že pole musí být vyhodnoceno manuálně, pomocí monitorovacího klienta.

Každé pole může mít rozdílný model validace. Některá pole v sobě mohou obsahovat speciální makra. Následuje nejprve seznam podporovaných maker a poté výčet typů polí a jejich atributů.

Makra

Součástí některých textových atributů mohou být výrazy, které je potřeba nahradit předem neznámou hodnotou. Pomocí této metody je možné kontrolovat hodnoty, jenž jsou zjistitelné pouze za běhu. Existujícími makry jsou:

- dest_mac:[ip_v4_adresa] – tento text bude nahrazen MAC adresou počítače se specifikovanou IPv4 adresou

  Příklad: {dest_mac:[192.168.1.1]} → 00:00:5e:00:53:af

- host_mac[:[iface]] – tento text bude nahrazen MAC adresou hostitelova počítače na rozhraní *iface* nebo na výchozím rozhraní

  Příklad: {host_mac:[eht2]} → 00:1b:44:11:3a:b7

  Příklad: {host_mac} → 00:1b:44:15:3a:af

- host_ip[:[*iface*]] – tento text bude nahrazen IPv4 adresou hostitelova počítače na rozhraní *iface* nebo na výchozím rozhraní

  Příklad: {host_ip:[eth5]} → 172.16.17.33

  Příklad: {host_ip} → 192.168.1.254

- host_ip_v6[:[*iface*]]:(link_local|global_unicast|multicast)[:no] – tento text bude nahrazen jednou IPv6 adresou hostitelova počítače možných typů:

  - link_local

  - global_unicast
  - multicast

  s číslem *no*, na rozhraní *iface* nebo na výchozím rozhraní.

  Příklad: {host_ip_v6:[eth4]:link_local} → fe80:0:0:0:0:0:0:1

  Příklad: {host_ip_v6:mutlicast:2} → ff02:0:0:0:0:0:1:2

- pc_num – text bude nahrazen hodnotou proměnné prostředí PC_NUM

  Příklad: {pc_num} → 5; systémová proměnná PC_NUM=5

Pole

Následující jsou programem podporovaná pole.

Text

```
field type: Text
validator: "{host_ip:[eth4]}"
```

Validní hodnotou pole *validator* je jakýkoli text. Tento text je vyhodnocen jako regulární výraz. Do tohoto textu lze dosadit makra.

Checkbox

```
field type: Checkbox
options:
  - Toto je odpověď číslo 1
  - Toto je odpověď číslo 2, dokonce s makrem čísla počítače {pc_num}
  - Toto je špatná odpověď
  - A toto také
validator: [0, 1]
```

Pole *options* je seznam libovolných textů s makrem.

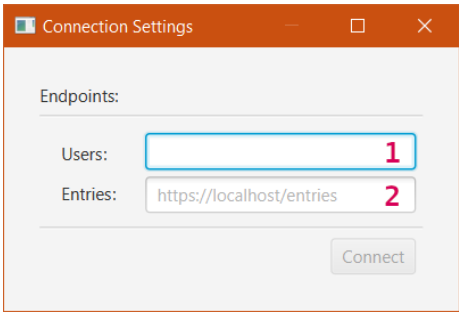
Pole *validator* je seznam čárkou oddělených správných odpovědí přímo související s polem *options*, indexování začíná od nuly (první možnost má index 0).

Monitorovací klient

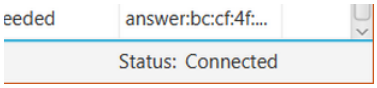
Tento klient je určen cvičícím předmětu, kteří potřebují kontrolovat stav studentů. Pro to, aby bylo možné použít tohoto klienta, je nutné mít spuštěnou serverovou část aplikace.

Postup:

1) Spusťte aplikaci.
2) V horní nabídce zvolte Connection > Connect.



3) Do kolonek 1 a 2 zadejte názvy koncových bodů pro uživatele, respektive záznamy serverové části aplikace. Musí se jednat o validní webové URL.
4) Stiskněte tlačítko Connect.
5) V prohlížeči se vám po několika vteřinách zobrazí stránka autorizačního serveru https://auth.fit.cvut.cz/, zde vyplňte své přihlašovací údaje a klikněte na tlačítko přihlásit.



6) Pokud nenastala žádná chyba, v pravém spodním okraji hlavního okna je nyní možné vidět **Status: Connected**.

Pokud se nezobrazují žádné záznamy, je pravděpodobné, že žádný zatím nebyl odeslán. Chcete-li obnovit tento seznam, vyberte z hlavního menu Data > Refresh.

Jakmile se zobrazuje dostatečný počet dat, že se zobrazí posuvník tabulky, nová data se načtou v okamžiku, kdy jej přesunete dolů (vyčerpáte počet zobrazených záznamů). Takto se data budou dále načítat, dokud není jejich zdroj kompletně vyčerpán.

Konfigurace monitorovacího serveru

Konfigurace monitorovacího serveru je uložena v souboru *application.yaml* adresáře `./monitoring-client/`. Součástí tohoto souboru je mnoho nastavení, které jde měnit bez potřeby kompilace. Následující seznam pouze shrnuje některé skupiny klíčů.

spring.application.datasource je určen pro nastavení databázového spojení, toto nastavení je potřeba změnit tak, aby souhlasilo s nastavením vašeho databázového systému. Bez nutnosti rekompilace je poskytnut pouze řadič pro systém MariaDB.

spring.security.oauth2 jsou záznamy týkající se autorizace, v případě změny adresy autorizačních serverů FIT ČVUT, případně usermapy je potřeba změnit tyto údaje. Druhým důvodem změny může být výměna klientů.

spring.security.oatuh2.resourceserver.usermap.educator-authorities je seznam všech rolí v rámci usermap ČVUT, které mají právo ohodnocovat (měnit) cizí záznamy nebo si cizí záznamy prohlížet.

server jsou obecné konfigurace serveru, například port, pod kterým server běží nebo nastavení certifikátů.

Příloha C

Koncové body API

Serverová aplikace prezentuje několik koncových bodů, ze kterých lze získat informace o testovacích záznamech.

Cílový bod – Entries

GET	/entries	Vrátí všechny záznamy

Formát odpovědi

```
"content": [{
    "id": 1,
    "username": "usr00001",
    "module": 0,
    "block": 0,
    "no": 1,
    "status": "Manual",
    "answer": "My answer",
    "timestamp": "2022-04-14T14:23:12.12001235+01:00"
}, …],
…
```

Pokud má odpověď vrátit pouze jediný záznam, nebude entita obalena polem content.
Pokud záznam neobsahuje odpověď, bude pole vynecháno.

Argumenty

username	Omezí záznamy na specifického uživatele.
module	Omezí záznamy na specifické číslo modulu.
block	Omezí záznamy na specifické číslo bloku.
no	Omezí záznamy na specifické číslo otázky/testu.
logicalId	Omezí záznamy na specifické číslo entity MBN
from	Omezí záznamy od tohoto data. Musí být specifikováno ve formátu ISO 8601.
	Výchozí hodnota: 1970-01-01T00:00:00Z
to	Omezí záznamy do tohoto data. Musí být specifikováno ve formátu ISO 8601.
	Výchozí hodnota: 9998-12-31T23:59:59.99999999Z
page	Omezí výpis na specifickou stránku (indexovanou od nuly)
	Výchozí hodnota: 0
size	Stanoví velikost stránky výpisu

	Výchozí hodnota: 20

Validní kombinace

username, from, to, page, size	Získá všechny záznamy uživatele.
Návratové kódy 403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam. 200 Ok – Jindy	
module, from, to, page, size	Získá všechny záznamy ze stanoveného modulu.
Návratové kódy 403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam. 200 Ok – Jindy	
username, module, from, to, page, size	Získá všechny záznamy uživatele ze stanoveného modulu.
Návratové kódy 403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam. 200 Ok – Jindy	
module, block, from, to, page, size	Získá všechny záznamy se stanoveným modulem a blokem.
Návratové kódy 403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam. 200 Ok – Jindy	
username, module, block, from, to, page, size	Získá všechny záznamy uživatele se stanoveným modulem a blokem.
Návratové kódy 403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam. 200 Ok – Jindy	
logicalId, from, to, page, size	Získá všechny záznamy stanovené entity MBN.
Návratové kódy 403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam. 200 Ok – Jindy	
username, module, block, no	Získá záznam uživatele se specifikovaným modulem, blokem a číslem otázky/testu.
Návratové kódy	

403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam.
404 Not Found – Pokud záznam se specifikovaným záznamem neexistuje.
200 Ok – Jindy

username, logicId	Získá záznam uživatele s danou MBN entitou

Návratové kódy
403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam.
404 Not Found – Pokud záznam se specifikovaným záznamem neexistuje.
200 Ok – Jindy

GET	/entries/{id}	Získá záznam s *id*

Návratové kódy
403 Forbidden – Když uživatel nemá oprávnění prohlížet daný záznam.
404 Not Found – Pokud záznam se specifikovaným záznamem neexistuje.
200 Ok – Jindy

POST	/entries	Vytvoří nový testovací záznam

Formát zprávy

```
{
    "username": "usr00001",
    "module": 0,
    "block": 0,
    "no": 0,
    "status": "Manual",
    "answer": "My answer"
}
```

Pole *answer* je volitelné.
Pole status má následující validní hodnoty:

- Success
- Manual
- Failed

Návratové kódy
400 Bad Request – Pokud status není validní hodnota.
403 Forbidden – Když se pokoušíte vytvořit záznam jiného uživatele.
404 Not Found – Pokud není nalezen specifikovaný uživatel.
409 Conflict – Pokud záznam se specifikovanými údaji již existuje.
201 Created – Pokud je záznam úspěšně vytvořen.

PUT	/entries/{id}	Nahradí testovací záznam

Formát zprávy

```
{
    "status": "Failed",
    "answer": "My answer"
}
```

Pole *answer* je volitelné.

Pole status má následující validní hodnoty:
- Success
- Manual
- Failed

Návratové kódy
400 Bad Request – Pokud status není validní hodnota.
403 Forbidden – Když se pokoušíte vytvořit záznam jiného uživatele.
404 Not Found – Pokud záznam s *id* neexistuje.
200 Ok – Pokud byl záznam úspěšně modifikován.

Cílový bod – Answers

Tento koncový bod je identický s bodem entries, ale zaručuje, že všechny odpovědi budou mít přítomné pole *answer*.

Cílový bod – Users

GET	/users	Vrátí seznam uživatelů

Formát odpovědi

```
"content": [{
   "username": "usr00001",
   "firstName": "Jméno",
   "lastName": "Příjmení",
   "middleName": "Prostřední"
}, …]
…
```

Pokud uživatel nemá prostřední jméno, bude pole vynecháno.

Argumenty

page	Omezí výpis na specifickou stránku (indexovanou od nuly)
	Výchozí hodnota: 0
size	Stanoví velikost stránky výpisu
	Výchozí hodnota: 20

GET	/users/{username}	Vrátí uživatele s daným uživatelským jménem

Formát odpovědi

```
{
   "username": "usr00001",
   "firstName": "Jméno",
   "lastName": "Příjmení",
   "middleName": "Prostřední"
}
```

Pokud uživatel nemá prostřední jméno, bude pole vynecháno.

<table>
<tr><td>GET</td><td>/users/{username}/entries</td><td>Vrátí záznamy uživatele</td></tr>
<tr><td colspan="3">Chování totožné s GET /entries?username=username</td></tr>
<tr><td>POST</td><td>/users/{username}/entries</td><td>Vytvoří záznam pro uživatele
s uživatelským jménem username.</td></tr>
<tr><td colspan="3">Formát zprávy
<pre>{
    "module": 0,
    "block": 0,
    "no": 0,
    "status": "Manual",
    "answer": "My answer"
}</pre>Pole answer je volitelné.
Pole status má následující validní hodnoty:
• Success
• Manual
• Failed
Návratové kódy
400 Bad Request – Pokud status není validní hodnota.
403 Forbidden – Když se pokoušíte vytvořit záznam jiného uživatele.
404 Not Found – Pokud není nalezen specifikovaný uživatel.
409 Conflict – Pokud záznam se specifikovanými údaji již existuje.
201 Created – Pokud je záznam úspěšně vytvořen.</td></tr>
<tr><td>PUT</td><td>/users/{username}/entries/{id}</td><td>Vytvoří nebo nahradí záznam s id
MBN entity pro uživatele
s uživatelským jménem username.</td></tr>
<tr><td colspan="3">Formát zprávy
<pre>{
    "status": "Failed",
    "answer": "My answer"
}</pre>Pole answer je volitelné.
Pole status má následující validní hodnoty:
• Success
• Manual
• Failed
Návratové kódy
400 Bad Request – Pokud status není validní hodnota.
403 Forbidden – Když se pokoušíte vytvořit záznam jiného uživatele.
404 Not Found – Když MBN entita se specifikovaným id neexistuje.
200 Ok – Pokud byl záznam úspěšně modifikován nebo vytvořen.</td></tr>
</table>

Příloha C

Cílový bod – Aggregates

Formát odpovědí

```
{
   "id": 1,
   "module": 0,
   "block": 0,
   "no": 1
}
```

GET	/aggregates?module=module&block=block&no=no	Vrátí entitu se specifikovaným MBN s modulem *module*, blokem *block* a číslem testu/otázky *no*.
GET	/aggregates/{id}	Vrátí entitu MBN s *id*.