VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SPRÁVA VÝPOČETNÍCH ÚLOH V SYSTÉMU FITCRACK

ADMINISTRATION OF COMPUTING TASKS IN FITCRACK SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ADAM HORÁK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK HRANICKÝ**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Horák Adam**
Program: Informační technologie
Název: **Správa výpočetních úloh v systému Fitcrack**
Administration of Computing Tasks in Fitcrack System
Kategorie: Web

Zadání:

1. Seznamte se s architekturou a implementací nástroje Fitcrack.
2. Po konzultaci s vedoucím navrhněte rozšíření, které usnadní tvorbu a správu lámacích úloh, např. seskupení úloh do skupin/adresářů, pokročilé filtrování, šablony pro import/export úloh, klonování úloh, apod.
3. Navržené řešení implementujte a ověřte jeho funkčnost.
4. Zhodnoťte dosažené výsledky.

Literatura:

- HRANICKÝ Radek, ZOBAL Lukáš, RYŠAVÝ Ondřej and KOLÁŘ Dušan. Distributed Password Cracking with BOINC and Hashcat. *Digital Investigation*, vol. 2019, no. 30, pp. 161-172. ISSN 1742-2876.
- HRANICKÝ Radek, ZOBAL Lukáš, VEČEŘA Vojtěch a MÚČKA Matúš. The Architecture of Fitcrack Distributed Password Cracking System. FIT-TR-2018-03, Brno, 2018.
- Další dle dohody s vedoucím

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a část bodu 3

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hranický Radek, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 29. října 2019

Abstrakt

Cílem této práce je navrhnout a následně implementovat rozšíření rozhraní správy úloh pro administrační aplikaci systému Fitcrack. Jedná se o řešení pro obnovu hesel ze zašifrovaných souborů a hešů. Fitcrack funguje na základě distribuce výpočtů na několik počítačů a zahrnuje velké množství možností konfigurace úloh. Cílem rozšíření navrhovaných v této práci je usnadnit uživateli zadávání nových úloh do systému a také manipulaci s existujícími. Mimo jiné umožní úlohy dělit do skupin, spouštět jich více v dávkách nebo je zálohovat do souboru.

Abstract

The aim of this thesis is to design and implement extensions for the Fitcrack system management application. The system is used to crack hashes and recover passwords from encrypted files. It works by distributing computation tasks across multiple computers and includes many configuration options. The extensions outlined in this thesis are meant to aid the user in creating and working with jobs in the system. These include grouping jobs, batch running, and exporting to a file.

Klíčová slova

Fitcrack, lámání hesel, API, webová aplikace, uživatelské rozhraní

Keywords

Fitcrack, password cracking, API, web app, user interface

Citace

HORÁK, Adam. *Správa výpočetních úloh v systému Fitcrack*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Hranický

Správa výpočetních úloh v systému Fitcrack

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Radka Hranického. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . . . .
Adam Horák
27. května 2020

Poděkování

Rád bych poděkoval Ing. Radkovi Hranickému nejen za vedení mé práce, ale i za to, že mi umožnil přidat se k týmu Fitcracku a dal mi tak příležitost přispět k něčemu smysluplnému při věnování se tomu, u čeho mě nejvíce baví ztrácet nervy.

Obsah

Kapitola 1

Úvod

Každé uživatelské rozhraní by se mělo snažit pomoci uživateli k jeho cíli. U jednoduchých systémů je obvykle záměr se uživateli takříkajíc neplést do cesty. V případě složitějších systémů se mu naopak vyhnout nedá. Snaha takového rozhraní je ideálně uživateli nabídnout způsoby, jak mu práci se složitým systémem zjednodušit. Vytvořit pro něj zkratky. Usnadnit mu nebo úplně odbourat repetitivní činnosti. Nabízet mu to, co je v daném kontextu nejvíce relevantní. A v neposlední řadě mu umožnit přehledně nahlížet na data a organizovat je.

Tato práce se zaměřuje konkrétně na správu úloh v systému Fitcrack, který slouží k obnově hesel ze zašifrovaných souborů a hešů. Obnova se provádí rozdělením práce na několik počítačů, které řídí hlavní server. Existuje řada způsobů, jak heslo získat a každý z nich zahrnuje jiné součásti, které je třeba různě konfigurovat. Celý tento systém je řízený z administrační aplikace, která musí umožňovat spravovat každou drobnost v celém tomto soukolí.

Jsou to právě systémy jako Fitcrack, jejichž administrační rozhraní obsluhuje nespočet funkcí a monitoruje kvanta dat, kde jsou různé pomůcky pro uživatele nedocenitelné. V současné administraci se všechny úlohy nachází v jednom výpisu, který s přidáváním dalších úloh v průběhu času rychle ztrácí na přehlednosti. Uživatel je nucen úlohy vyhledávat napříč stránkami podle jména nebo stavu. Navíc systém neumožňuje pracovat s více úlohami současně, což by právě při udržování pořádku uživatel zajisté ocenil.

Pro zpříjemnění používání aplikace navrhuje tato práce několik rozšíření a dokumentuje jejich zavedení do systému. Jejich hlavním přínosem je lepší možnost organizovat úlohy a k nim přidružená data do logických celků a efektivněji s nimi pracovat. Další úpravy a sjednocení aplikace pak zjednodušují orientaci v jednotlivých sekcích. Rozsah úprav je podle povahy jednotlivých rozšíření od změn pouze ve webové aplikaci až po zcela nové funkční celky zasahující do celého systému administrace, systémové databáze a přidružených serverových modulů.

V průběhu sedmi kapitol se tento dokument věnuje seznámení se systémem Fitcrack, návrhu rozšíření administrace, jejich implementaci a testování. Kapitola 2 seznamuje s architekturou celého systému a popisuje fungování jednotlivých částí. Modul administrace, technologie na kterých stojí a současnou podobu správy úloh probírá kapitola 3. V kapitole 4 jsou popsány návrhy různých rozšíření pro administrační aplikaci. Je vysvětlen jejich záměr a nastíněn způsob jejich používání. Jejich následná implementace je postupně zdokumentována v kapitole 5. Kapitola 6 zaznamenává testování a experimenty provedené k ověření správného fungování nových součástí i celého systému. Závěrečná kapitola 7 pak shrnuje celé snažení a zhodnocuje výsledky.

Kapitola 2

Systém Fitcrack

Fitcrack je systém sloužící k obnově (lámání) hesel z hešů nebo z podporovaných souborů, jako jsou dokumenty nebo archívy. Vznikl jako nástupce nástroje Wrathion[1]. Ten dokázal lámání provádět na CPU nebo pomocí OpenCL[2] na grafických jednotkách, ovšem pouze na jednom stroji. Na rozdíl od něj je Fitcrack koncipovaný jako distribuovaný systém, který se dělí na serverovou část a klientské uzly k ní připojené. Původně, tak jako Wrathion, využíval vlastní implementace jádra v OpenCL. To bylo ovšem později nahrazeno lámacím nástrojem Hashcat, který je blíže popsán v části 2.2.

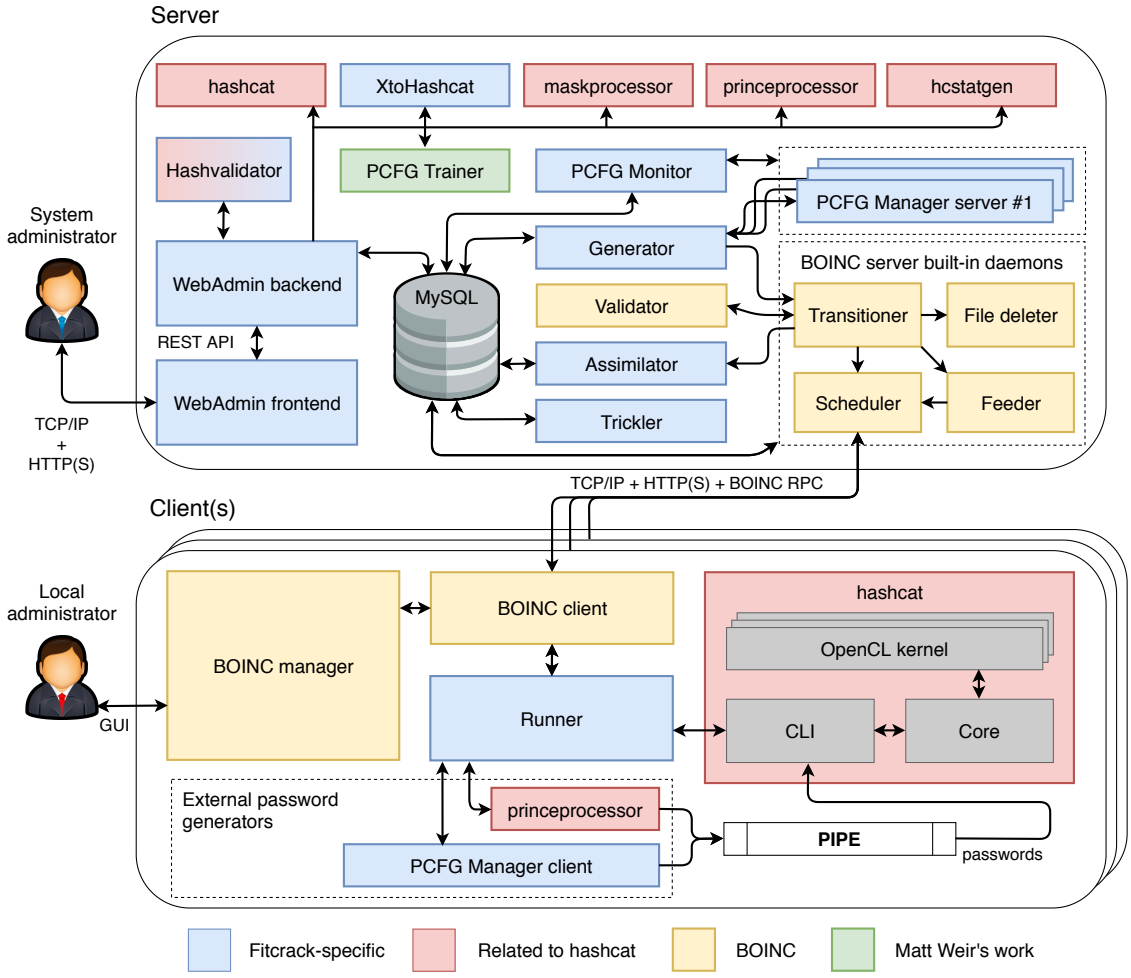
Server se skládá z několika součástí, které jsou blíže vysvětleny v části 2.3. Jejich úkolem je rozdělit zadanou lámací úlohu na menší části zvané workunit, které jsou pak přiděleny jednotlivým klientům. Tyto klientské uzly jsou k serveru připojeny v rámci frameworku BOINC popisovanému v části 2.1. Takových klientů může do systému být připojeno teoreticky neomezené množství. Tím je umožněno zadanou úlohu zpracovávat paralelně.

Klientské uzly mohou běžet na různých platformách a mohou být různě vybavené. Jedinou podmínkou pro připojení uzlu je, aby se na něm nacházel klientský modul BOINC. Ten komunikuje se serverem a přijímá nové úkoly. Server pak prostřednictvím tohoto modulu klientům předává vše potřebné pro práci na úkolech. Po zadání úkolu uzly pracují na lámání svých přidělených částí workunit a výsledky hlásí serveru. Ten je zpracovává a podle vyhodnoceného aktuálního stavu dává klientům další instrukce [8].

2.1 Framework BOINC

Právě BOINC umožňuje Fitcracku rozdělovat úlohy mezi připojené uzly. Na obrázku 2.1 jsou jeho součásti vyznačeny žlutě. BOINC (Berkeley Open Infrastructure for Network Computing)[3] je platforma pro distribuované výpočty. Je vyvíjena na Kalifornské univerzitě v Berkeley skupinou, která vyvinula SETI@home[4]. Jedná se o open source software. Na straně serveru sestává z plánovače a správce souborů. Plánovač rozhoduje, jak rozdělit jednotky workunit mezi připojené klienty. O přidělení práce si ovšem žádá každý klient sám podle svého lokálního plánovače. Důvodem je fakt, že BOINC je navržen pro distribuci výpočetních úloh mezi dobrovolníky. Počítá s tím, že kterýkoliv klient se může kdykoliv odpojit nebo vrátit nesprávný výsledek. Další z přidaných vlastností jsou takzvané „trickle messages“, tedy zasílání zpráv v průběhu práce na workunit. Jejich účelem je umožnit

[1] https://wrathion.fit.vutbr.cz/
[2] Open Computing Language – https://www.khronos.org/opencl/
[3] https://boinc.berkeley.edu
[4] https://setiathome.berkeley.edu



Obrázek 2.1: Schéma architektury systému Fitcrack [8]

dobrovolné klienty průběžně „odměňovat“ při práci na dlouho trvajících výpočtech, jako je předpověď počasí [1].

Spousta těchto funkcí BOINC nemá pro Fitcrack takový význam, protože se k němu typicky nepřipojují dobrovolníci z celého světa. Stále však přináší jisté výhody. Například trickle zprávy se používají k průběžnému hlášení stavu výpočtu na uzlu, což pak umožňuje tento zobrazovat v administračním rozhraní [7].

2.2 Nástroj Hashcat

Hashcat[5] je nástroj pro obnovu hesel, který pro výpočty využívá OpenCL. Je samozvaným nejrychlejším lámacím nástrojem. V současnosti podporuje přes 220 kryptografických algoritmů, pět různých typů útoku a několik druhů zařízení podporujících OpenCL. Aktuálně jej využívá systém Fitcrack jako program pro samotné lámání hesel, který je spouštěn na jednotlivých uzlech. Na obrázku 2.1 jsou jeho součásti vyznačeny červeně[6].

[5] https://hashcat.net/hashcat/

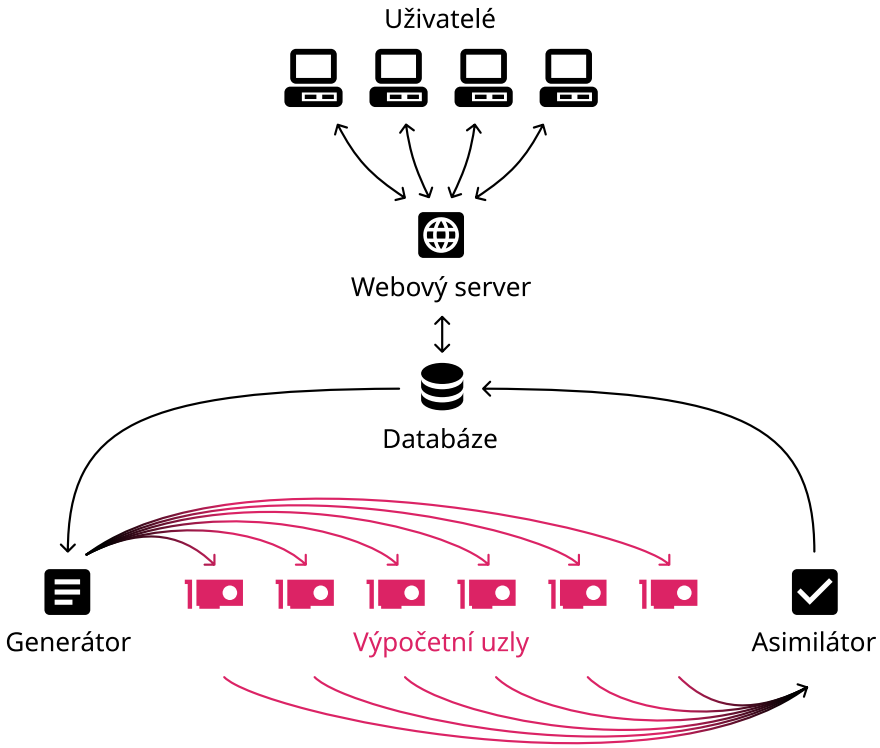
[6] Hashvalidator je vlastní implementace založená převážně na zdrojovém kódu z nástroje Hashcat [8]

XtoHashcat

Program Hashcat pracuje pouze se samotnými řetězci hešů. K jejich extrakci ze zašifrovaných souborů slouží různé skripty třetích stran. Pro zjednodušení práce uživatele systému Fitcrack byl vyvinut nástroj XtoHashcat, který tyto skripty zapouzdřuje a dokáže automaticky rozpoznat typ souboru. Uživateli pak stačí soubor v rozhraní administrace nahrát a heš bude extrahován. V současnosti XtoHashcat podporuje dokumenty sady Microsoft Office, dokumenty PDF a archívy RAR, ZIP a 7z [8].

2.3 Architektura systému Fitcrack

V této části jsou přiblíženy součásti architektury systému Fitcrack. Zjednodušený pohled na celý systém, bez součástí frameworku BOINC a některých modulů, je znázorněn na obrázku 2.2.



Obrázek 2.2: Zjednodušené znázornění architektury systému Fitcrack [9]

2.3.1 Webový server

Webserver obsluhuje administrační část systému, které se dále věnuje kapitola 3. Pomocí webové aplikace se k němu mohou připojit uživatelé a pracovat se systémem – sledovat průběh úloh, zadávat nové, spravovat kolekce používané pro útoky, jako například slovníky, spravovat uživatele a další.

Systém administrace je vytvořený jako oddělený systém. V klientském webovém prohlížeči běží jednostránková aplikace, tedy statická webová stránka, která asynchronně, skrze

rozhraní REST (viz 3.1), komunikuje se serverem (backend). Ten pak s využitím dalších modulů obstarává požadavky zaslané aplikací, které vyvolal uživatel výše zmíněnými akcemi. Změny zapisuje do databáze a z té také čte aktuální stav, ovšem nekomunikuje přímo s dalšími součástmi systému. Databáze tak slouží jako prostředník mezi administračním modulem a zbytek systému Fitcrack.

2.3.2 Generátor

Generátor vytváří a rozděluje pro připojené klienty pracovní jednotky workunit. Také se vypořádává se situacemi, kdy se klient odpojí nebo vrátí chybný výsledek. Snaží se práci rozdělit mezi jednotlivé uzly tak, aby každá přidělená část odpovídala výkonu uzlu. K tomu je potřeba výkon každého uzlu změřit a proto generátor vytváří dva typy jednotek:

Benchmark

Benchmark workunit slouží k měření výkonu výpočetního uzlu. Po prvotním připojení uzlu do systému pro něj vytvoří generátor takzvaný kompletní benchmark, který změří výkon uzlu při práci na všech typech úloh. Tento výsledek je pak zaznamenán v databázi a dále využíván pro odhady času potřebného k lámání.

Běžná úloha

Běžně zasílané jednotky obsahují informace o samotném zadání práce pro uzel. Podle typu útoku generátor připraví potřebné soubory, které jsou s jednotkou zaslány. Kolik hesel má v rámci jednotky být vyzkoušeno je odvozeno od výkonu uzlu naměřeného benchmark workunity a složitosti použitého kryptografického algoritmu.

2.3.3 Výpočetní uzly

Výpočetní uzly jsou klientské počítače, které jsou v rámci frameworku BOINC připojené k serveru systému Fitcrack. Na nich běží program Runner, který stojí nad nástrojem Hashcat a slouží k jeho řízení. Při spuštění BOINC klientem přečte konfigurační soubor zaslaný serverem a podle něj nastaví a spustí lámání v nástroji Hashcat. V průběhu úlohy ji monitoruje a pak vytváří výstupní soubory, které opět předá BOINC klientu.

2.3.4 Asimilátor

Asimilátor přijímá zprávy od jednotlivých uzlů. Přijaté výsledky zpracovává a zapisuje do databáze. Mohou to být zprávy o dokončené jednotce benchmark nebo o výsledku lámání. Pokud některý uzel nalezne heslo, pak asimilátor upraví záznam v databázi mimo jiné tak, že nastaví stav úlohy na dokončeno. Ostatním uzlům je pak předána informace o tom, že již mohou výpočty ukončit.

2.3.5 Další moduly

Systém sestává i z mnoha dalších modulů, jak je možné vidět v diagramu na obrázku 2.1. Jedná se o součásti frameworku BIONC, podpůrné moduly pro obsluhu nástroje hashcat a specializované součásti pro podporu některých typů útoku. V systému se také nachází několik služeb běžících na pozadí (daemon), které periodicky kontrolují stav systému nebo

změny informací v databázi. Pomáhají tak informovat uživatele o aktuálním dění v systému a automaticky obsluhují podpůrné programy.

Moduly serveru BOINC

Server frameworku BOINC využívá několika k němu dodávaných modulů. Feeder vyhrazuje úsek sdílené paměti pro předávání informací ve formě záznamů z databáze. Transitioner zpracovává přechody mezi stavy pracovních jednotek. Podle potřeby vytváří výsledky a rozpoznává chyby. Validator ověřuje výsledky získané od výpočetních uzlů. File Deleter maže na serveru soubory se zadáním a další prostředky, které již nejsou potřeba [1].

Pomocné moduly serverové části administrace

Serverová část systému administrace využívá zejména při zpracování zadání úloh různé podpůrné součásti a programy. Výpočet velikostí množin hesel a další informace poskytuje přímo nástroj hashcat. Nástroj Hashvalidator ověřuje správnost zadaných hešů. XtoHashcat umožňuje extrahovat heše ze souborů, blíže je popsán v části 2.2. Nástroj hcstatgen vytváří ze slovníků soubory statistik (Markov Chains).

Podpůrné součásti pro útoky PCFG a PRINCE

Lámání pomocí útoků PCFG a PRINCE využívá externích nástrojů. Útok PCFG využívá bezkontextové gramatiky, které vytváří nástroj PCFG Trainer s využitím slovníků. Pro každou lámanou úlohu pak instance nástroje PCFG Manager generuje kandidáty hesel z těchto gramatik [6]. Spouštění a správu instancí programu PCFG Manager zajišťuje služba PCFG Monitor. Pro generování hesel pomocí algoritmu PRINCE pak slouží nástroj princeprocessor.

Kapitola 3

Administrace systému Fitcrack

K ovládání systému Fitcrack slouží modul administrace, složený ze statické, jednostránkové webové aplikace a API[1] serveru s několika dalšími podpůrnými moduly. Takto navržený web umožňuje úplně oddělit vývoj a provoz obou částí a snadněji je také testovat. Také lze přes výsledné API obsluhovat více aplikací, pokud by to bylo v budoucnu potřeba. Cenou za tuto flexibilitu je ovšem celkově složitější vývoj celého systému a nutná koordinace mezi týmy pracujícími na projektu. Je třeba předem naplánovat a dohodnout podobu komunikačního rozhraní mezi jednotlivými součástmi. Při rozšiřování systému je pak nutná spolupráce při návrhu, aby rozhraní zůstalo jednotné a nová funkcionalita fungovala správně na všech vrstvách. V této kapitole jsou dále popsány obě části administrace a technologie, na kterých stojí.

3.1 Serverová část

Serverová část primárně zpracovává požadavky klienta a zapisuje do databáze, která je středem komunikace celého systému. Navíc obsahuje nástroj Hashcat, stejný program, který je používán na výpočetních uzlech k lámání hesel. Ten je zde využíván pro podporu některých funkcí v administraci, jako je výpočet rozsahu množiny možných hesel (tzv. keyspace). Server také vyřizuje požadavky na extrakci heše ze zašifrovaných souborů pomocí nástroje XtoHashcat (viz 2.2). Další součásti, které server využívá, jako hcstatgen nebo princeprocessor, jsou popsány v části 2.3.5. Server systému administrace je postavený na frameworku Flask v jazyce Python.

Python

Skriptovací jazyk Python[2] vytvořil a v roce 1991 vydal Guido van Rossum. Je to dynamicky typovaný interpretovaný jazyk zaměřený na jednoduchost a čitelnost kódu. Je objektově orientovaný, ale podporuje i další paradigmata [12].

[1] Application Programming Interface – https://en.wikipedia.org/wiki/Web_API
[2] https://www.python.org

Framework Flask

Webový framework Flask[3] je jeden z nejpoužívanějších nástrojů pro tvorbu webových serverů v Pythonu. Je postaven jako mikroframework, který se podle potřeby dále rozšiřuje moduly.

Knihovna SQLAlchemy

SQLAlchemy[4] je sada nástrojů pro práci s relační databází pro jazyk Python. Implementuje nad konkrétní databází abstrakci ve formě vrstvy ORM[5].

Architektura REST

Representational State Transfer (REST) je architektura jednotného přístupu ke zdrojům rozmístěným v síti. Navrhl ji Roy Fielding v roce 2000 ve své dizertační práci [4]. Popisuje komunikaci klienta se serverem, která je bezstavová, tedy taková, kde každý požadavek sám o sobě obsahuje všechny podstatné informace pro jeho vyřízení a na serveru se neukládá žádná informace o stavu klienta. Komunikace probíhá pomocí sloves protokolu HTTP, kdy se požadavky zasílají na konkrétní místo určené unikátním identifikátorem URI[6].

3.2 Klientská aplikace

Klient administrace je jednostránková webová aplikace, tedy statická stránka, ve které se při navigaci nenačítá ze serveru celý nový dokument, ale pouze se asynchronně přenesou data samotného požadovaného obsahu. Prezentační logika stránky pak překreslí jen části, které je třeba vyměnit nebo změnit jejich textový obsah. V případě aplikace pro Fitcrack toto zajišťuje knihovna Vue.

Asynchronní přenos dat

Přenos dat prováděný skriptem na stránce bez nutnosti načítat a vykreslovat úplně nový dokument. Tato technika je jeden ze základů moderních webových stránek a aplikací, ať už jsou jednostránkové nebo ne. Původně byl řešený v rámci rozšíření jako byl Flash, ale časem se staly potřebné technologie součástí prohlížečů a vžil se souhrnný název AJAX[7]. Lze takto stahovat a posílat libovolná data, ale nejčastější využití je přenos dat ve formátu JSON[8] za účelem dynamické aktualizace prezentace na straně klienta. JSON je odvozen ze standardu ECMAScript a je určen k výměně dat nezávisle na jazyku [3].

Vue

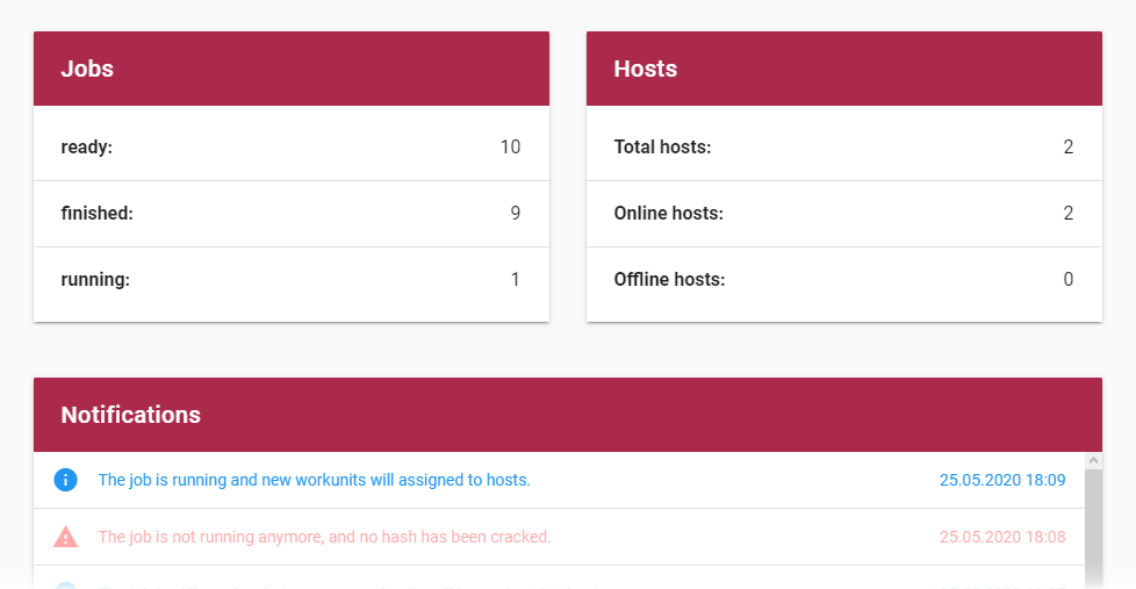
Vue[9] (vue.js) je knihovna pro vytváření webových rozhraní, kterou původně vytvořil Evan You. Podobně jako některé další frameworky a knihovny se zaměřuje na abstrakci reaktivity – umožňuje automaticky aktualizovat zobrazení (vykreslené šablony) při změně datové

[3] https://www.palletsprojects.com/p/flask/
[4] https://sqlalche.me
[5] Object-Relational Mapping
[6] Uniform Resource Identifier
[7] Asynchronous JavaScript and XML – https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX
[8] JavaScript Object Notation – https://www.json.org/
[9] https://vuejs.org

struktury pod ním. Funkcionalita i vzhled se pak zapouzdřují do komponent, které lze skládat do větších celků. Tak uživatel vytvoří kompletní aplikaci. Vue navíc disponuje spoustou oficiálních rozšíření, jako je Vue Router pro směrování v aplikacích [13].

3.3 Rozhraní pro správu úloh

Vytváření, správa a monitorování lámacích úloh jsou jedny z hlavních funkcí administrace. Na hlavní stránce aplikace se nachází dashboard, tedy přehled nejdůležitějších údajů z celého systému. Jeho aktuální podoba je na obrázku 3.1.



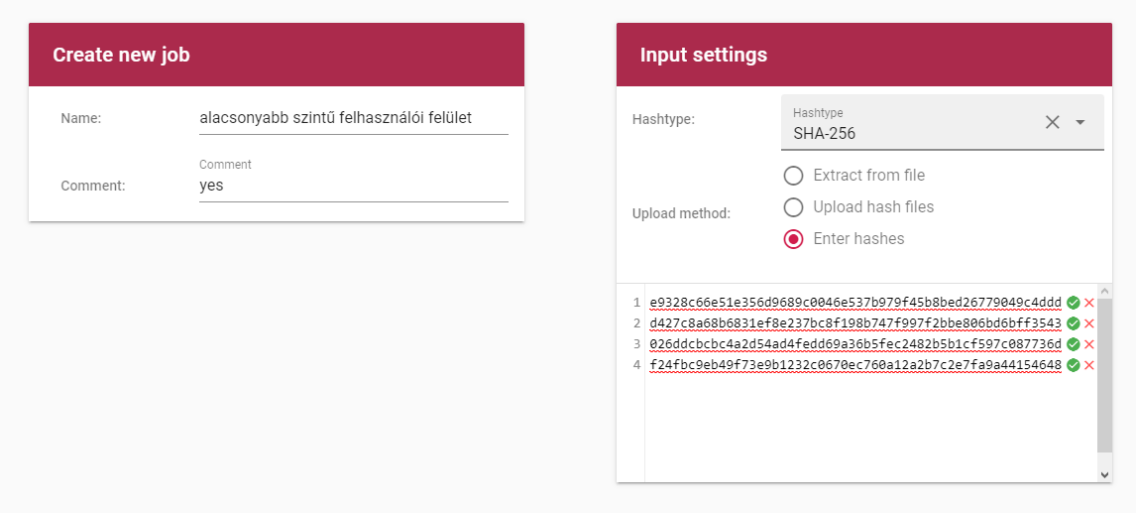
Obrázek 3.1: Přehled o systému v administrační aplikaci

Mimo jiné je zde i přehled, v jakých stavech se nachází úlohy a kolik jich je. Dále výpis stavu několika úloh, vybraných podle poslední aktivity. Ve spodní části se pak nachází grafy zobrazující celkovou aktivitu úloh a uzlů v systému.

3.3.1 Přidávání úlohy

Zadávání úlohy do systému je jedna z nejkomplexnějších částí celé administrační aplikace. Uživateli je potřeba prezentovat několik sekcí nastavení, která se navíc mohou lišit v závislosti na vybraném typu útoku. V nejnovější verzi aplikace je tento proces rozdělen do čtyř kroků:

- **Zadání vstupu** – zde uživatel zadává heše, které se bude úloha pokoušet prolomit. Zadané hodnoty jsou průběžně kontrolovány konzultací se serverem, kde se ověřuje jejich validita. Zadání je možné přímo do textového pole, z textového souboru nebo extrakcí (viz 2.2). Na obrázku 3.2 je příklad vstupu.
- **Nastavení útoku** – v tomto kroku uživatel vybere, jaký typ útoku se má v úloze použít. V závislosti na tom se zobrazují různé ovládací prvky, které umožňují do detailu nastavit parametry.



Obrázek 3.2: Formulář zadání vstupu pro úlohu

- **Přiřazení výpočetním uzlům** – tento krok umožňuje vybrat, na které výpočetní uzly připojené aktuálně do systému se bude úloha distribuovat. Ve výchozím stavu jsou vybrány všechny dostupné uzly.
- **Dodatečná nastavení** – tato sekce nabízí možnost naplánovat spuštění úlohy na konkrétní čas, omezit dobu jejího běhu, nastavit čas pro jednotlivé pracovní jednotky a v neposlední řadě přidat k celé úloze poznámku.

Během zadávání se přímo v aplikaci kontroluje správnost vybraných voleb. Pokud některé povinné volby chybí nebo některé nejsou kompatibilní, nelze úlohu vytvořit.

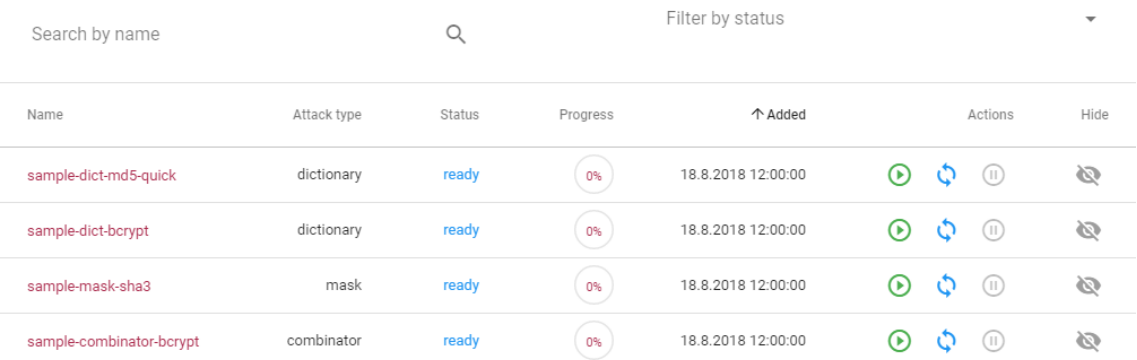
3.3.2 Přehled úloh v systému

Na stránce přehledu úloh jsou ve stránkované tabulce vypsány veškeré úlohy evidované v systému a nejdůležitější informace o nich. Zachycena je na obrázku 3.3. Tabulku lze řadit podle většiny sloupců a lze v ní vyhledávat podle názvu úlohy. Filtrování je umožněno pouze na základě stavu úlohy.

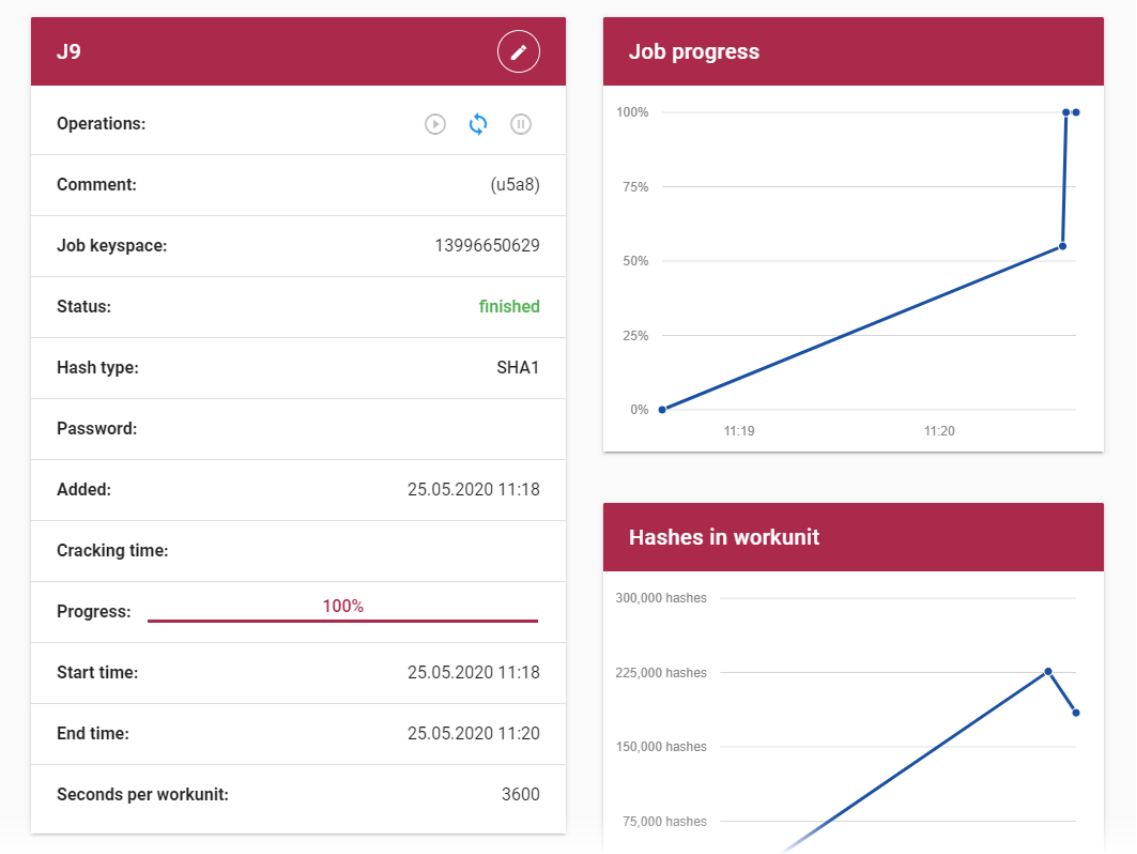
Součástí tabulky jsou také rychlé akce, které umožňují úlohu přímo ovládat bez nutnosti otvírat detailní zobrazení. Jednou z těchto akcí je skrytí úlohy. Takto skryté úlohy jsou pak k nalezení v oddělené tabulce, která se dá zobrazit pomocí přepínače nad tabulkou.

3.3.3 Detail úlohy

Detailní zobrazení úlohy obsahuje několik modulů rozdělených do karet na stránce. Horní část zobrazuje údaje o úloze nastavené při vytvoření a statistiky průběhu a rozdělení práce, včetně grafů. Tato část je k vidění na obrázku 3.4. Lze zde také některá nastavení upravit, například rozdělení mezi uzly nebo název a plány úlohy. Spodní část pak obsahuje tabulku s detaily o jednotlivých pracovních jednotkách. Ke každému řádku v tabulce je přidružen i výpis záznamu z vykonávání pracovní jednotky na uzlu.



Obrázek 3.3: Tabulka úloh evidovaných v systému



Obrázek 3.4: Část detailního přehledu úlohy

Kapitola 4

Návrh rozšíření

V této kapitole se budu věnovat návrhu rozšíření a úprav administračního systému pro Fitcrack. Jedná se obvykle o doplňující funkcionalitu, která nemá za cíl přidat do systému nové funkce jako takové, ale urychlit a pokud možno zpříjemnit používání těch existujících.

Na úvod

Ve své bakalářské práci se autor původní implementace systémových analytik a dashboardu věnuje možným rozšířením. Často se jedná o možnost uživatele přizpůsobit si komponenty či celé rozhraní podle svého uvážení [11].

Některé z těchto bodů byly zapracovány do aplikace při její modernizaci, jak popisuje část 4.1. Pokud jde o prostorovou úspornost rozhraní, zde si dovolím nesouhlasit. Přílišná snaha co nejvíce maximalizovat informační hustotu nevede na přehlednost [5]. Grafické uživatelské rozhraní potřebuje splnit určité zásady, aby se uživateli takříkajíc nepleto do cesty a naopak mu bylo přínosem. Mimo jiné bych mezi ně zařadil:

- zobrazování relevantních informací,
- smysluplné rozložení prvků,
- rozmístění prvků s dostatečným odstupem.

Celá aplikace je postavena na frameworku implementujícím Material Design[1], ovšem na spoustě míst je využito stylů, které jsou s ním v rozporu a to hlavně z hlediska právě odstupu mezi jednotlivými prvky. Během modernizace rozhraní jsem se snažil více respektovat zásady systému Material Design a stejně tak budu činit i při návrhu rozhraní pro má rozšíření.

4.1 Modernizace rozhraní aplikace

V zájmu modernizování celé prezentace aplikace a jako základ pro má dále popisovaná rozšíření aktualizuji použitý framework uživatelského rozhraní, Vuetify[2], na verzi 2. Ta implementuje aktualizovaný standard Material Design 2 a také přináší spoustu nových funkcí

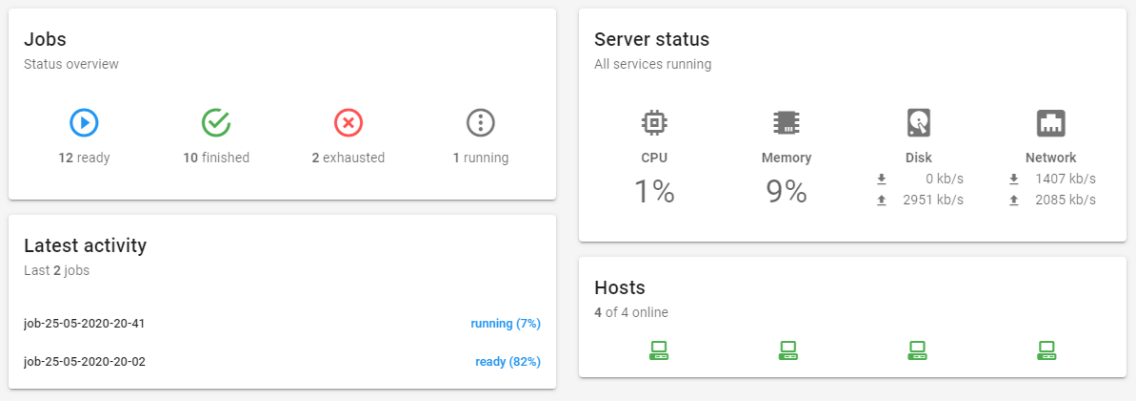
[1] https://material.io
[2] https://vuetifyjs.com/

a změny aplikačního rozhraní. Z toho důvodu je třeba velkou část dosavadní webové aplikace přizpůsobit těmto změnám. Při této příležitosti jsem i přepracoval spoustu komponent aplikace, aby lépe ladily s novou podobou celého produktu. Některé větší změny jsou popsány částech následujících, jako zcela nová stránka úlohy (část 4.8) nebo výpis úloh (část 4.6).

Nová verze Vuetify také zjednodušuje přizpůsobení barevné palety a umožňuje dynamicky měnit motiv celé aplikace. Díky tomu může Fitcrack respektovat předvolbu motivu v operačním systému a uživatel si může zvolit světlý nebo tmavý vzhled. I z toho důvodu jsem se rozhodl zmírnit v aplikaci využití primární barvy z palety jako pozadí. Velké plochy se omezí na neutrální barvy a zbytek palety poslouží pouze jako akcenty pro ovládací prvky a důležité ukazatele.

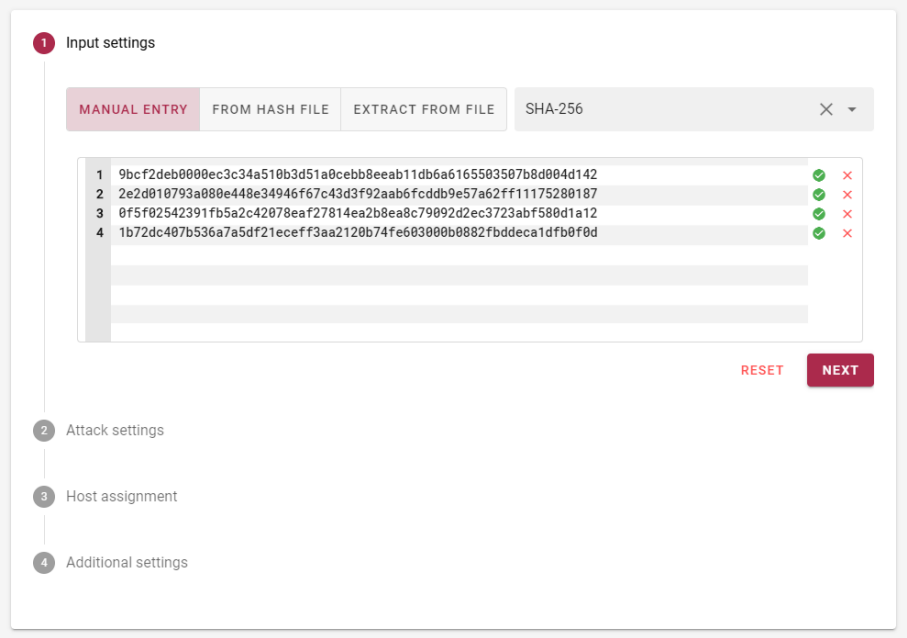
Nové rozhraní pro dashboard a formulář zadání úlohy

Výraznější změny se dotknou rozhraní hlavní stránky (dashboard) a formuláře pro zadávání nové úlohy do systému. Některé informace zobrazované na dashboardu je možné konsolidovat a podat intuitivněji „v jedné větě“. Orientaci mezi jednotlivými kartami a prvky také usnadní více barevně odlišených hodnot a širší využití piktogramů. Ukázka takto upraveného přehledu je na obrázku 4.1. Oproti původnímu řešení (viz obrázek 3.1) je zde v kontextu rychlého přehledu nad systémem k vidění více informací i přes využití velkých symbolů a méně explicitně uvedených hodnot.



Obrázek 4.1: Nová podoba dashboardu

Formulář zadávání úlohy se skládá z mnoha částí, které je třeba vyplnit. Jejich rozdělením do několika kroků vzniknou logické celky s jasnějším účelem. Uživatel je bude procházet postupně a budou aktivní pouze jednotlivě. Tak nebude uživatel nucen se posouvat dlouhou stránkou, když některé volby může přeskočit nebo jim ponechat výchozí hodnotu. Příklad formuláře ve stavu, kdy uživatel vyplňuje vstupní heše na začátku zadávání je na obrázku 4.2. Zbytek formuláře je zde skrytý pod následujícími kroky. Ve srovnání s původní podobou formuláře, jak je vidět i z části zadávání zachycené na obrázku 3.2, dostanou také prvky potřebný horizontální prostor.



Obrázek 4.2: Zadávání vstupu pro úlohu ve formuláři se skrytým zbytkem kroků

4.2 Ukládání a nahrávání šablon

Šablony úloh umožní uživateli předvyplnit formulář pro zadávání úlohy do systému (viz 3.3.1). Většina voleb dostupných ve formuláři má potenciál být v znovu použita v určité kombinaci jako základ pro vytváření nové úlohy.

Jedinou výjimkou jsou vstupní heše úlohy, které bych do nastavení úlohy ani nepočítal. Jejich ukládání v rámci pouhých šablon by navíc přineslo zbytečné technické komplikace. V rámci práce se šablonami úloh je potřeba vyřešit tři diskrétní potřeby – jejich vytváření, aplikaci a správu.

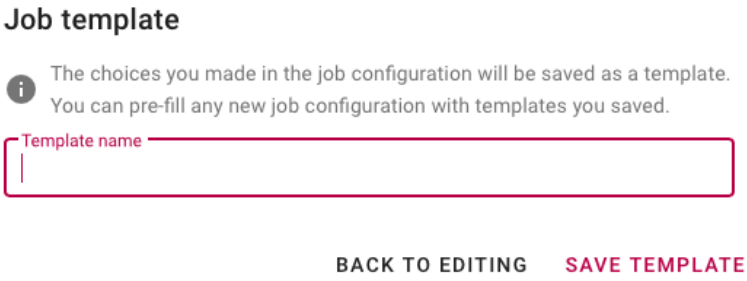
Vytvoření šablony

Pro vytváření šablon bude použit samotný formulář zadávání úloh. Uživatel zadá údaje stejně, jako při zadávání úlohy. Tlačítkem vytvoření šablony vyvolá dialog (viz obrázek 4.3), kde pouze potvrdí nebo změní jméno pro šablonu. Po vytvoření šablony uživatel zůstane ve formuláři a může jej dále upravovat nebo hotovou úlohu zadat do systému. Na straně serveru se přijatá konfigurace šablony ve formátu JSON uloží do databáze pro následné načítání při její aplikaci.

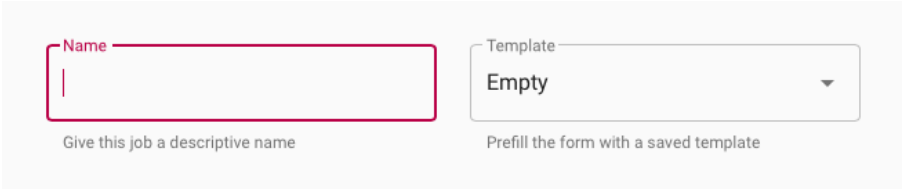
Aplikace šablony

Aby se možnost předvyplnění formuláře stala snadněji objevitelnou, bude vhodné systém šablon představit jako jeden z klíčových prvků formuláře a ne jen jeho nadstavbu. Šablony se tedy budou používat vždy. I prázdný formulář se tak stane šablonou, která bude ve výchozím stavu vybrána. Zároveň tak přirozeně vznikne možnost dosavadní konfiguraci smazat a začít znovu.

Výběr šablony, která se má použít, bude jedna z prvních možností ve formuláři (viz obrázek 4.4), aby si uživatel co nejdříve vybral základ, na kterém bude dále pokračovat v kon-



Obrázek 4.3: Dialog vytvoření šablony



Obrázek 4.4: Výběr šablony vedle vstupního pole názvu

figuraci. Protože šablon může v systému časem přibýt nesčetně mnoho, ideálním vstupem pro jejich výběr bude vyhledávací pole s automatickým doplňováním existujících šablon.

Protože šablony ovlivňují data ve spoustě prvků vstupního formuláře, bude potřeba ukládat aktuální konfiguraci jinak, než přímo v komponentách formuláře. Pokud bude využito úložiště stavu aplikace, bude možnost zároveň vyřešit jeden z neduhů současného systému: ztrátu veškeré doposud zadané konfigurace při navigaci mimo formulář. K tomuto dochází poměrně často, protože formulář na spoustě míst odkazuje na jiná místa v aplikaci, jako jsou detaily o různých souborech, které lze v konfiguraci využít.

Správa existujících šablon

Pro správu existujících šablon postačí jednoduché zobrazení seznamu s možností náhledu šablony, jejího přejmenování a samozřejmě smazání. Samostatné rozhraní pro úpravu šablony vzhledem k pravděpodobně malému využití takové možnosti není prioritou. Jelikož lze šablonu jednoduše nahrát do formuláře a uložit, postačí zavést unikátní názvy. Úprava šablony tak bude fungovat formou nahrání a následného uložení šablony pod stejným názvem, kdy dojde k jejímu přepsání.

4.3 Třídění úloh do skupin

Dělení úloh do skupin umožní uživateli libovolně sdružovat úlohy do pojmenovaných pohledů a provádět nad nimi hromadné operace. V současné aplikaci lze za podobnou, leč nesrovnatelně primitivnější, funkci považovat skrývání úloh. Skryté úlohy se dají zobrazit v tabulce zvlášť a tak by se daly považovat za skupinu úloh.

Oproti skrývání bude seskupování velice flexibilní a mělo by skrývání úplně nahradit. K tomuto účelu bude sloužit speciální oddělená skupina, odpadkový koš. Cílem je vytvořit místo, kam uživatel „zahodí“ již nepotřebné úlohy, což byl nakonec i jeden z hlavních

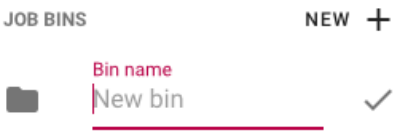
případů užití funkce skrývání. V současném systému nelze úlohy z administrace skutečně mazat a ani k tomu není důvod, pokud zůstane zachována tato možnost skrytí úlohy v koši.

Seskupování by se mělo stát primární součástí aplikace. Jádro funkcionality skupin bude zasazené do hlavního navigačního panelu na straně aplikace. Aby byl k tomuto účelu dobře použitelný, dostane navigační panel úplně nové rozložení, které dále probírá část 4.4.

Podobně jako u šablon celá funkcionalita seskupování sestává z několika diskrétních součástí – vytváření skupin, jejich úprava a uspořádávání, přidávání a odebírání úloh.

Vytváření a správa skupin

Vytvářet a uspořádávat skupiny bude moci uživatel přímo z navigačního panelu. V hlavičce části vyhrazené pro uživatelem definované skupiny bude tlačítko pro vytvoření skupiny. Po jeho stisku se v seznamu objeví nová skupina, kterou uživatel pojmenuje a následně potvrdí vytvoření. Toto znázorňuje obrázek 4.5. Uživatel bude mít samozřejmě možnost vytváření zrušit.



Obrázek 4.5: Vytváření nové skupiny úloh

K přesouvání skupin v seznamu pak poslouží gesto chycení a tažení (drag & drop). Přejmenování a mazání skupin jsou naopak akce, které bude vhodné umístit přímo do záhlaví zobrazení konkrétní skupiny. Návrh na obrázku 4.6 navíc v záhlaví zobrazuje celkový počet úloh ve skupině.

Obrázek 4.6: Záhlaví skupiny s dodatečnými akcemi

Přesouvání úloh mezi skupinami

Pro účely přesouvání úloh přibude v tabulce možnost výběru řádků zaškrtávacím políčkem. Pokud bude vybrán alespoň jeden řádek, zobrazí se v navigačním panelu u každé skupiny volba přesunout vybrané úlohy tam. U aktuálně otevřené skupiny bude naopak možnost vybrané úlohy vyřadit. Protože je navigační panel vždy dostupný vedle tabulky, není potřeba pro tyto akce implementovat oddělené rozhraní.

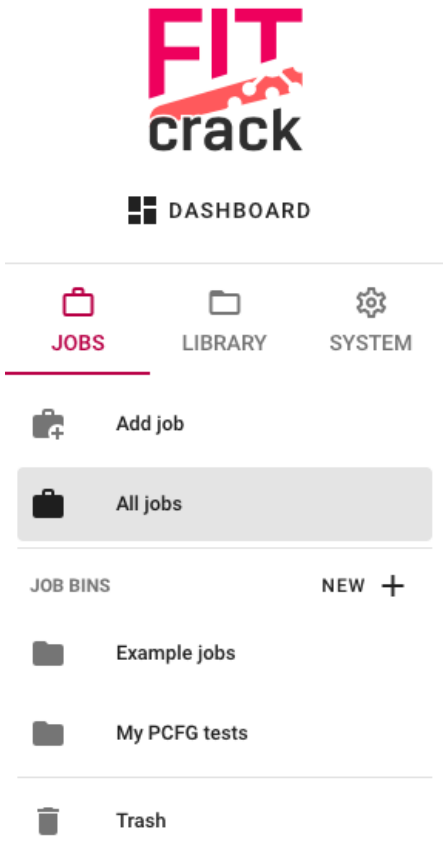
Jedna úloha může být přiřazena do více skupin. Zobrazení všech úloh, které aktuálně v administraci existuje, bude vždy obsahovat všechny úlohy v systému, kromě těch přesunutých do koše.

4.4 Navigační panel

Z důvodu integrace skupin úloh do navigačního panelu a stálého potenciálu aplikaci rozšiřovat o nové sekce jsem pro něj potřeboval navrhnout úplně nové rozhraní. Namísto prostého umístění všech odkazů pod sebe bude nový panel rozdělený do tří záložek:

- **Záložka úloh** – část věnující se pouze úlohám v systému, uvolňující prostor pro zobrazení skupin
- **Záložka knihovny** – část pro všechny spravovatelné podpůrné součásti systému, jako slovníky, pravidla, masky, apod.
- **Záložka systému** – část zahrnující sekce pro správu systému, připojených uzlů, uživatelů a také sledování prostředků

Návrh rozložení je zachycen na obrázku 4.7. Hlavní sekce, tedy dashboard, bude dostupná vždy nad samotnými záložkami. Protože se panel může vždy posouvat vertikálně, lze do něj umístit teoreticky nekonečně mnoho skupin úloh nebo jiných odkazů. Takto by mohl fungovat i nerozdělený, ovšem za cenu radikálně snížené přehlednosti po přidání spousty skupin.



Obrázek 4.7: Rozdělený navigační panel

4.5 Hromadné operace nad úlohami

Ve spoustě situací by bylo vhodné uživateli nabídnout možnosti, jak ovlivnit parametry více existujících úloh současně. Ne u všech nastavitelných vlastností či operací je ale vhodné toto zavádět. Současné provedení některých operací by totiž mohlo v některých případech negativně ovlivnit stabilitu nebo i ohrozit funkčnost celého systému.

Nejproblematičtější v tomto ohledu jsou operace spouštění, zastavování a restartování úloh. Proveditelnost těchto operací je závislá na aktuálním stavu každé z úloh. I když se nabízí řešení v podobě provedení zvolené operace nad všemi úlohami z vybraných, kde je to možné, působí toto spíše jako matoucí a nedomyšlený systém. Namísto toho navrhuji zavést systém dávkového spouštění, který je blíže specifikován v části 4.7.

Akce, které navrhuji zavést pro zvolené položky ve výpisu úloh, jsou následující. Samozřejmě lze k těmto zařadit i akci přidávání či odebírání úloh z uživatelem vytvořených skupin, které je představeno v části 4.3.

- Přiřazení výpočetních uzlů k více úlohám,
- přesun více úloh do koše a zpět.

4.6 Sjednocený výpis úloh

Pro skupiny úloh bude potřeba v podstatě totožný výpis jako pro zobrazení všech úloh v systému. Navíc má tento výpis ve všech podobách nabýt nových možností, jako jsou zmiňované hromadné operace. Rozhodl jsem se proto využít tuto příležitost k celkové modernizaci tohoto zobrazení. Vedle přidávání funkcionality a přizpůsobování pro výpis úloh ze skupin jsem se zaměřil i na zlepšení celkové přívětivosti této komponenty.

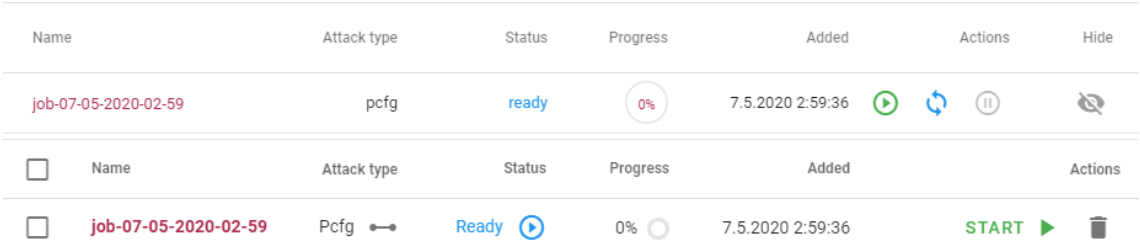
Přizpůsobení pro výpis skupin

Ve své podstatě je výpis úloh ve skupině úplně stejný, jako výpis všech úloh v systému. Proto bude vhodné znovu použít stejnou komponentu. Aby ale bylo možné provádět akce jako přejmenování nebo smazání skupiny, jak nastiňuje část 4.3, bude zaveden návrh na obrázku 4.6 jako záhlaví pro všechny výpisy. V případě, že se vypisuje skupina, bude záhlaví obsahovat tlačítka pro zmiňované úpravy. Nadpis pohledu společně s ukazatelem počtu úloh pak bude figurovat v každé variaci pro zlepšení orientace.

Úprava tabulky s daty

Samotná tabulka vypisující data o jednotlivých úlohách neumožňuje výběr položek, který bude potřeba pro hromadné akce a třídění. Tuto funkcionalitu doplní zaškrtávací pole v každém řádku tabulky a v záhlaví. Celá tabulka je navíc poměrně nepřehledná, s jednolitým textem, který je odlišitelný jedině barvou. Odkazy na úlohy jsou rovněž omezené aktivní plochou na samotný text. Bude vhodné některé prvky zdůraznit a doplnit ikonami zavedenými v novém formuláři pro zadávání a dashboardu. Přístupnost pak může zlepšit i možnost kliknutí na odkaz i v oblasti okolo textu.

Ve sloupci s akcemi pro každou úlohu se nachází skupina tlačítek pro spuštění, restartování a zastavení. V závislosti na stavu úlohy se pak jednotlivá tlačítka aktivují a deaktivují. Při bližším zkoumání jsem ale došel k závěru, že v množině stavů, ve kterých se může úloha nacházet, neexistuje žádný, který by tlačítek vyžadoval více najednou. Proto může skupina



Obrázek 4.8: Srovnání řádku tabulky před (nahoře) a po úpravě (dole)

tlačítek být nahrazena pouze jedním, které je aktuálně potřeba. Obrázek 4.8 porovnává původní řádek s novým návrhem. Chování pak bude odpovídat tabulce 4.1.

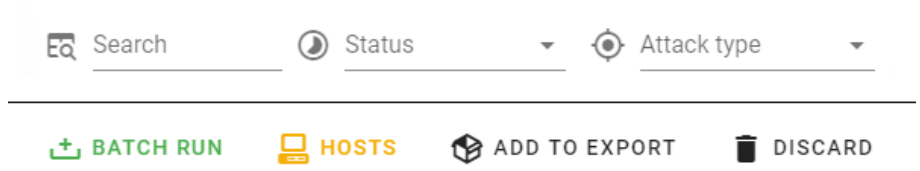
Stav úlohy	Operace
Připravená	Start
Běží nebo se dokončuje	Stop
Skončila	Restart

Tabulka 4.1: Proveditelná operace podle stavu úlohy

Ve sloupci stavu bude navíc nově uživatel upozorněn na chybějící přiřazení výpočetního uzlu k úloze. Tento stav zároveň ovlivní operační tlačítka. Úloha, která je připravená, ale nemá přiřazený žádný uzel, nepůjde spustit.

Kombinovaný panel nástrojů

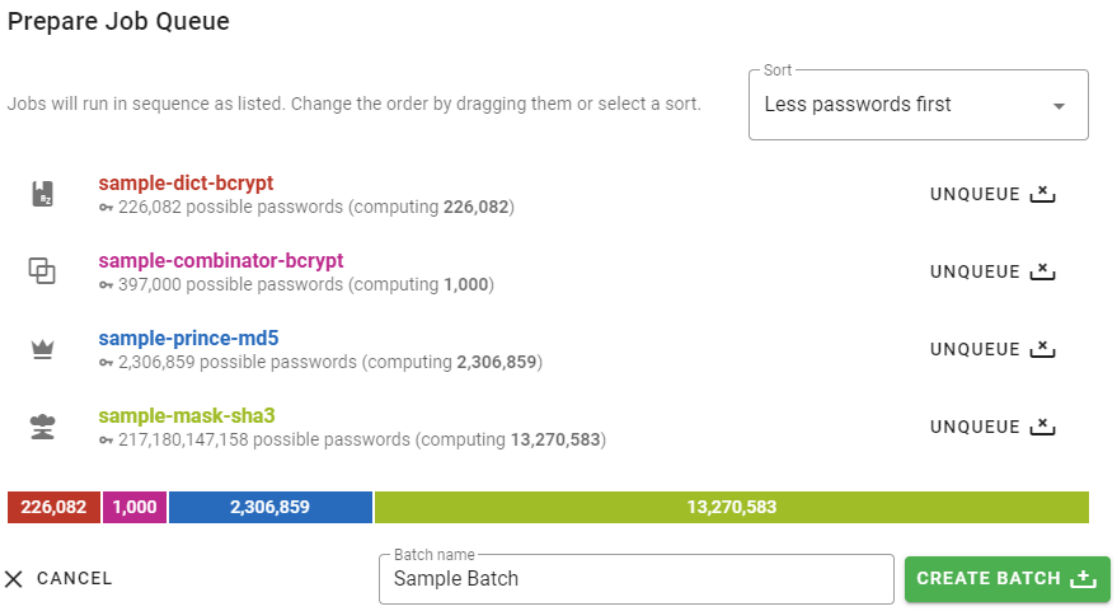
Panel vyhledávání a filtrování nově umožní filtrovat úlohy nejen podle stavu, ale i podle typu útoku. Navíc ale dostane druhou funkci. Pokud uživatel vybere alespoň jednu úlohu pomocí zaškrtávacích polí v tabulce, změní se obsah panelu na skupinu tlačítek pro hromadné akce, viz část 4.5. Zde pak uživatel najde veškeré akce spojené s vybranými úlohami. Tak vznikne i prostor pro další budoucí rozšíření, které by mělo takto vybrané úlohy ovlivňovat, protože uživatel bude zvyklý hledat je na tomto panelu. Oba režimy panelu jsou na obrázku 4.9.



Obrázek 4.9: Panel nástrojů v režimu filtrování (nahoře) a výběru (dole)

4.7 Dávkové spouštění úloh

Fitcrack jako systém staví na distribuci výpočetních úloh mezi připojené uzly. Ideálně má tedy každá úloha k dispozici všechny uzly najednou a může se mezi ně efektivně rozdělit zátěž v podobě pracovních jednotek. Pokud by ovšem uživatel v současnosti chtěl spustit více úloh, nemá moc možností, jak zachovat tento ideální stav. Systém totiž nemá pojem



Obrázek 4.10: Dialog vytvoření dávky

o tom, která úloha by měla mít přednost, protože k implementaci priorit nikdy nedošlo. Uživatel tak má v podstatě jen tyto možnosti:

- spustit více úloh a spokojit se s nepředvídatelným generováním pracovních jednotek,
- přiřadit úlohám ručně unikátní výpočetní uzly.

Ani jedno řešení není ideální. Pro zachování efektivnosti systému bude mnohem vhodnější úlohy spouštět postupně v sekvenci. Proto navrhuji systém dávek, díky kterému bude uživatel moci explicitně zadat, které úlohy chce spustit za sebou a v jakém pořadí.

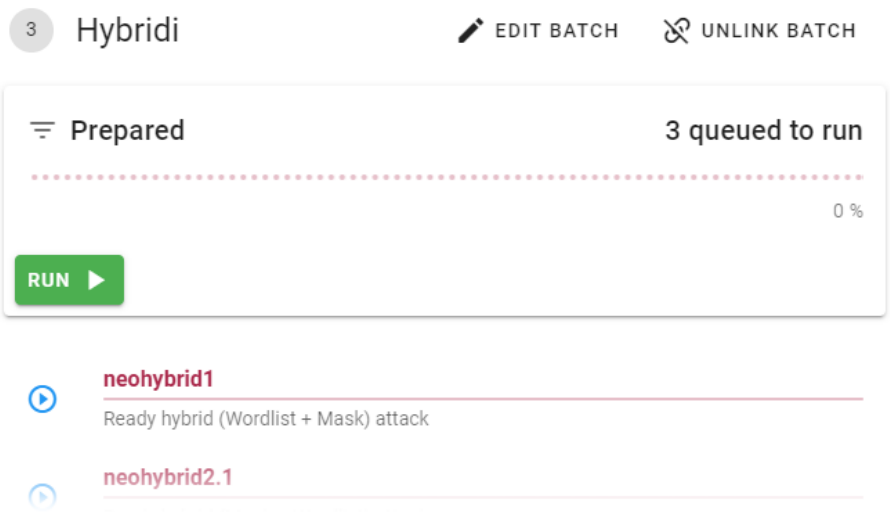
Vytvoření dávky

Pro vytvoření dávky nejprve uživatel v libovolném výpisu úloh vybere ty úlohy, které mají být součástí dávky. Platné jsou pouze úlohy, které jsou připravené a ještě nebyly spuštěny (stav *ready*). Tlačítkem *Batch Run* na panelu nástrojů (viz obrázek 4.9) se otevře dialog s náhledem fronty úloh, jak ukazuje obrázek 4.10.

Úlohy v dialogu může uživatel libovolně přetahovat na jiné místo a přizpůsobit si tak pořadí, ve kterém budou spouštěny. Může si také úlohy nechat seřadit podle určitých parametrů, jako je počet možných hesel nebo náročnost výpočtu. Ve spodní části pak uvidí vizualizaci fronty, kde velikost každého dílku naznačuje rozsah úlohy a tedy i předpokládaný poměr dob potřebných k jejich dokončení.

Správa dávek

Po zadání se dávka přidá k ostatním do výpisu dávek. Odtud si může uživatel zobrazit detail jednotlivých dávek (obrázek 4.11), kde vidí aktuální stav celé fronty úloh, aktuální průběh a společné statistiky. Ty ukazují rozdělení práce všech úloh ve frontě mezi všechny zúčastněné výpočetní uzly a objemy pracovních jednotek vzhledem k průběhu lámání.



Obrázek 4.11: Detail dávky

Dávky bude možné přejmenovat a také zrušit, tedy je odstranit ze seznamu a rozdělit vazby mezi úlohami. Spouštění a zastavování lámání se v dávkách bude řídit samotnými úlohami. Stav celé dávky bude odrážet stav právě aktivních úloh, případně úlohy, která je jako první na řadě.

4.8 Přepracovaný detail úlohy

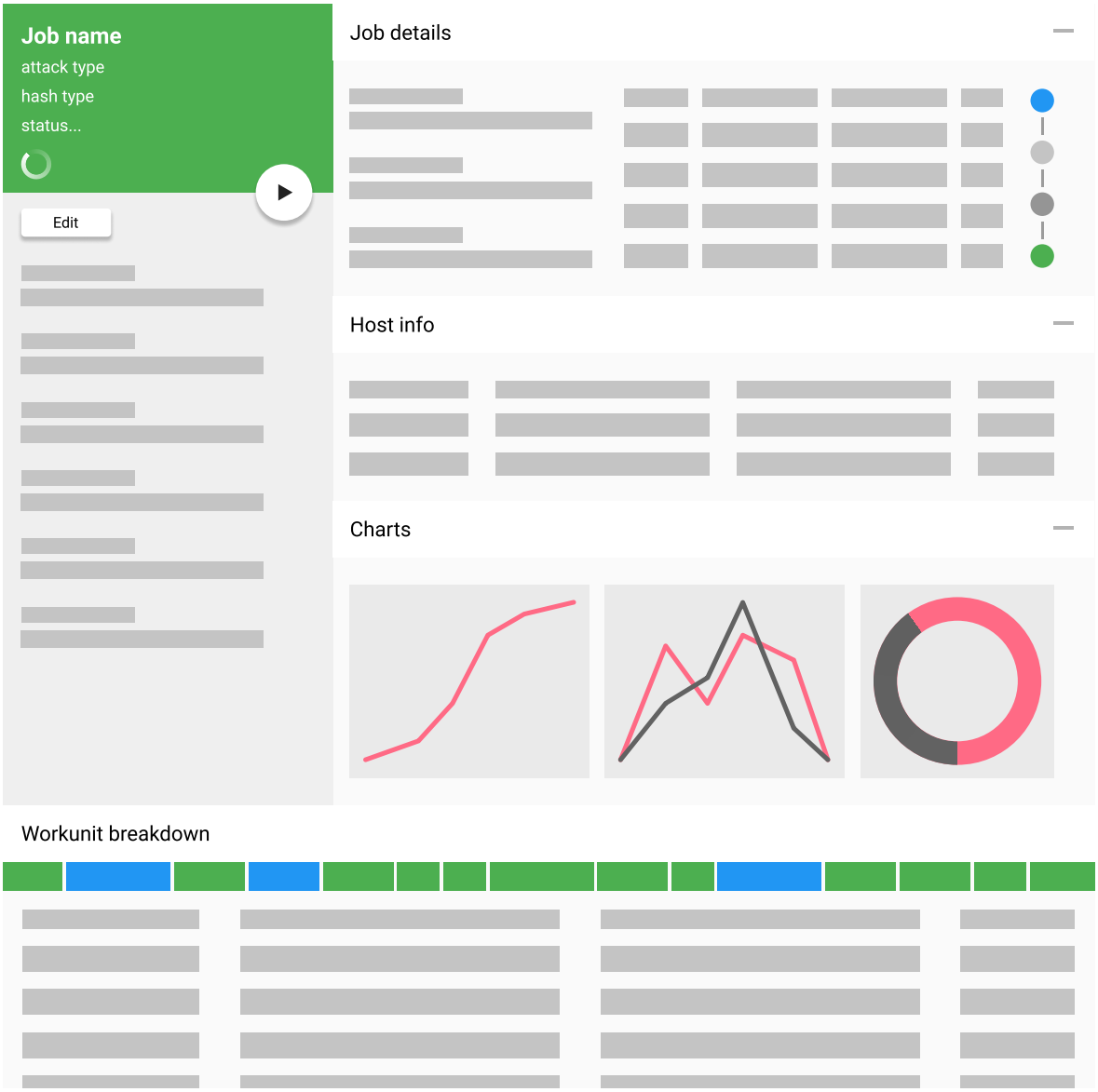
S postupným přidáváním funkcí do systému a rozšiřováním možností úloh se také rozrostl počet komponent zobrazovaných v detailu úlohy. Rozhraní je zde rozděleno do dvou sloupců, v nichž jsou všechny komponenty rozmístěny zdánlivě náhodně. Ve spodní části je pak detail pracovních jednotek, který vyplňuje celou šířku stránky.

I když jsem do tohoto pohledu nepřidával nic, co by vyžadovalo větší změny rozložení, chci jej přepracovat do přehlednější podoby s logičtějším umístěním prvků. V dosavadním rozložení je potřeba často stránku vertikálně posouvat a tak lze snadno ztratit přehled. Navíc grafy vykreslující statistiky úlohy narušují responzivitu sloupců, protože se jejich velikost mění skriptem a nereagují na zmenšování v modelu flexbox[3].

Nová stránka bude také rozdělena do dvou sloupců, ovšem jeden z nich bude určen pro základní informace o úloze a její ovládání. Obsah tohoto sloupce je v původním rozložení zahrnut v jedné z karet jako prostý seznam, který byl na širších obrazovkách zbytečně roztažený a špatně čitelný. Namísto toho, dokud bude vedle něj dostatek místa, bude mít sloupec fixní šířku. V horní části bude na první pohled vše podstatné o úloze, jako název, typ útoku a heše, či průběh.

Vedlejší sloupec pak bude zahrnovat zbytek informací, rozdělený do kategorií, které lze jednotlivě otevřít nebo minimalizovat. Takto může v budoucnu být přidáno více komponent při zachování přehlednosti nad základními informacemi. Ve spodní části pak zůstává rozbor pracovních jednotek v plné šířce. Náhled návrhu stránky detailu poskytuje obrázek 4.12.

[3] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout

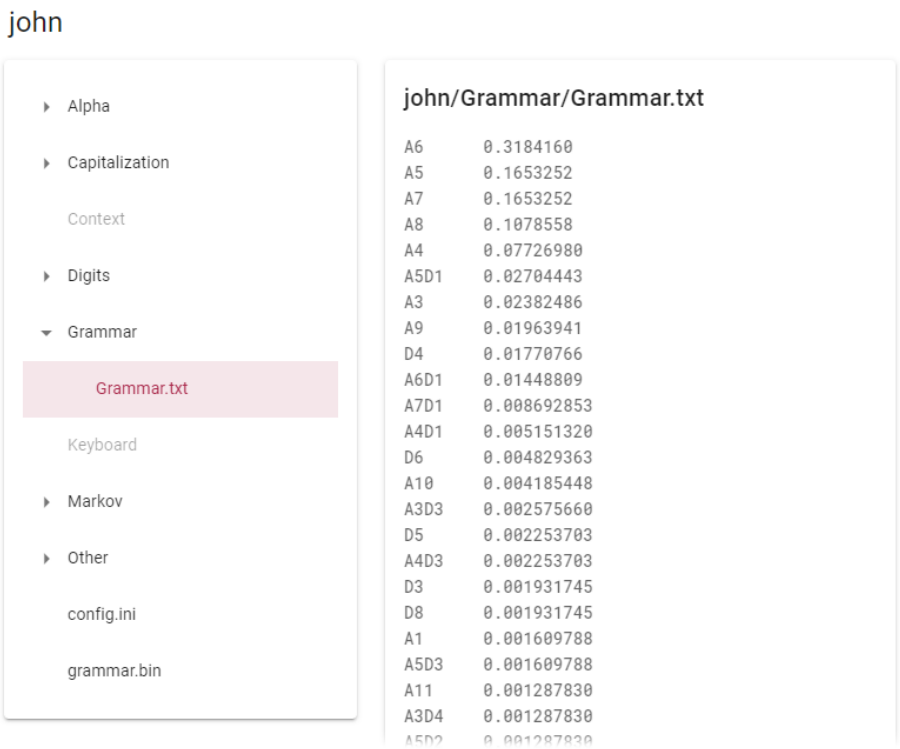


Obrázek 4.12: Návrh detailního zobrazení úlohy

4.9 Prohlížeč součástí PCFG

Pro většinu prostředků používaných k lámání, jako jsou například slovníky nebo znakové sady, je v administrační aplikaci dostupný náhled obsahu. Jedná se obvykle o jeden textový soubor, který lze snadno uživateli vypsat. Pro gramatiky PCFG, které se skládají z mnoha souborů v adresářové struktuře, tento náhled chybí. Tato podoba adresářové struktury a obsažených souborů je používána, protože jí ve stejné formě očekává nástroj PCFG Trainer, viz část 2.3.5.

Můj návrh na stránku náhledu pro PCFG sestává ze dvou částí: interaktivního stromu adresářů a samotného výpisu obsahu vybraného souboru. Strom zobrazí obsah kořenového adresáře gramatiky. Pokud je položka stromu adresář s obsahem, může jej uživatel otevřít a dále procházet. Takto mohou být adresáře zanořeny do teoreticky neomezené hloubky. Pokud je položka soubor, pak se jeho výběrem vypíše obsah v panelu pro výpis. Příklad prohlížeče je na obrázku 4.13.



Obrázek 4.13: Otevřený soubor v prohlížeči PCFG

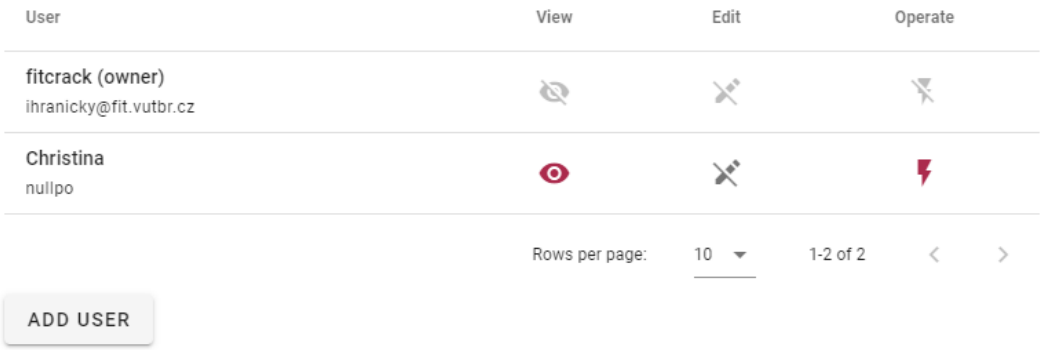
4.10 Pokročilý systém oprávnění

Již v původním návrhu administračního systému pro Fitcrack se počítalo se systémem oprávnění, který ovšem nikdy nebyl dokončen. V systému mají uživatelé přiřazenou roli, která jim uděluje určitá oprávnění. Role lze libovolně vytvářet a měnit a stejnou roli může mít více uživatelů. Oprávnění, která v systému již existují, i když z většiny nevyužitá, jsou:

- správa uživatelských účtů,
- vytváření úloh,
- zobrazení všech úloh v systému,
- úprava všech úloh v systému,
- ovládání všech úloh v systému,
- nahrávání nových slovníků.

Z těchto bylo doposud využíváno pouze omezení pro vytváření úloh, kdy se jen v aplikaci skrylo tlačítko s odkazem na danou stránku. Pro úplnou implementaci je třeba nejen přizpůsobit rozhraní aplikace, ale i zabezpečit koncové body nebo některé jejich akce na straně serveru.

Další vrstvou systému oprávnění má být možnost přidělovat uživatelům oprávnění k zobrazení, editaci a ovládání úlohy individuálně pro každou úlohu. K tomu bude sloužit dialog, zachycený na obrázku 4.14, který se stane součástí komponenty editace úlohy, představené v části 4.8. Pro minimalizaci komplikací jsem se rozhodl tuto činnost umožnit pouze uživateli, který úlohu vytvořil a který k ní má vždy veškerá oprávnění. Tohoto uživatele budu dále nazývat vlastníkem úlohy.



Obrázek 4.14: Dialog pro správu oprávnění konkrétní úlohy

Některá má rozšíření navíc souvisí s úlohami a tak se i jich týká tento systém oprávnění. Skupiny (viz 4.3) jsou globálně dostupné a obsahují úlohy, tudíž je potřeba omezit i jejich výpis. Oprávnění skupiny přesouvat a vytvářet pak bude zastřešovat oprávnění upravovat všechny úlohy.

Dávky (viz 4.7) sestávají z úloh, ovšem zde by bylo velice komplikované a pro uživatele matoucí, kdyby se mu zobrazovaly dávky podle toho, zda má oprávnění na zobrazení všech úloh v dávce. Uživatel by navíc například náhle ztratil možnost pozastavit dávku, pokud by právě běžící úlohu neměl právo ovládat. Racionálnější řešení je zavést i u dávek vlastníky

a zobrazovat každému uživateli pouze jeho dávky. Zobrazení všech by pak bylo umožněno tomu, kdo má oprávnění zobrazovat a ovládat všechny úlohy v systému.

4.11 Přenos dat mezi instalacemi

V systému Fitcrack neexistuje žádná možnost zálohovat nebo přenášet uživatelem vytvořené úlohy. V rámci tohoto návrhu chci umožnit uživateli vybrat konkrétní úlohy k zálohování a ty uložit do souboru. Vzhledem k tomu, že architektura systému nikdy nepočítala s hromadným exportováním a importováním úloh, bude nutné přistoupit na určité kompromisy.

Úlohy v systému jsou, podle jejich typu útoku, závislé na různých prostředcích, které mohou být v některých případech velmi objemné. Například slovníky, používané v několika různých typech útoku, mohou dosahovat velikostí v řádech gigabajtů. Přidávat tyto prostředky do souboru se zálohou je tak nepraktické a celý proces by to mohlo značně zpomalovat. Namísto toho budou zálohy pouze znát své závislosti na těchto prostředcích. Uživatel pak bude při obnovení zálohy požádán o jejich doplnění do cílového systému v případě, že tam chybí.

Pro výběr úloh, které se mají v záloze zahrnout, poslouží opět zaškrtávací pole ve výpisu úloh, jak popisuje část 4.6. Vybrané úlohy se pak přidají do seznamu pro export, který bude dostupný z nové části aplikace určené pro práci se zálohami. Zde bude mít uživatel možnost úlohy ze seznamu opět odebrat, pokud změní názor, a následně nechat vytvořit soubor se zálohou, který internetový prohlížeč stáhne.

Na stejné stránce pak bude možnost vybrat soubor k nahrání na server, z něhož se dříve uložená data importují. Před provedením importu se zobrazí informace o souboru a zkontroluje se přítomnost zaznamenaných závislostí na serveru. Rovněž se vypíše seznam úloh obsažených v souboru pro kontrolu uživatelem. V případě, že nechybí žádné součásti uvedené v závislostech, může uživatel úlohy ze souboru nechat přidat do systému. Náhled stránky ve stavu před importováním zálohy je na obrázku 4.15.

4.12 Další rozšíření

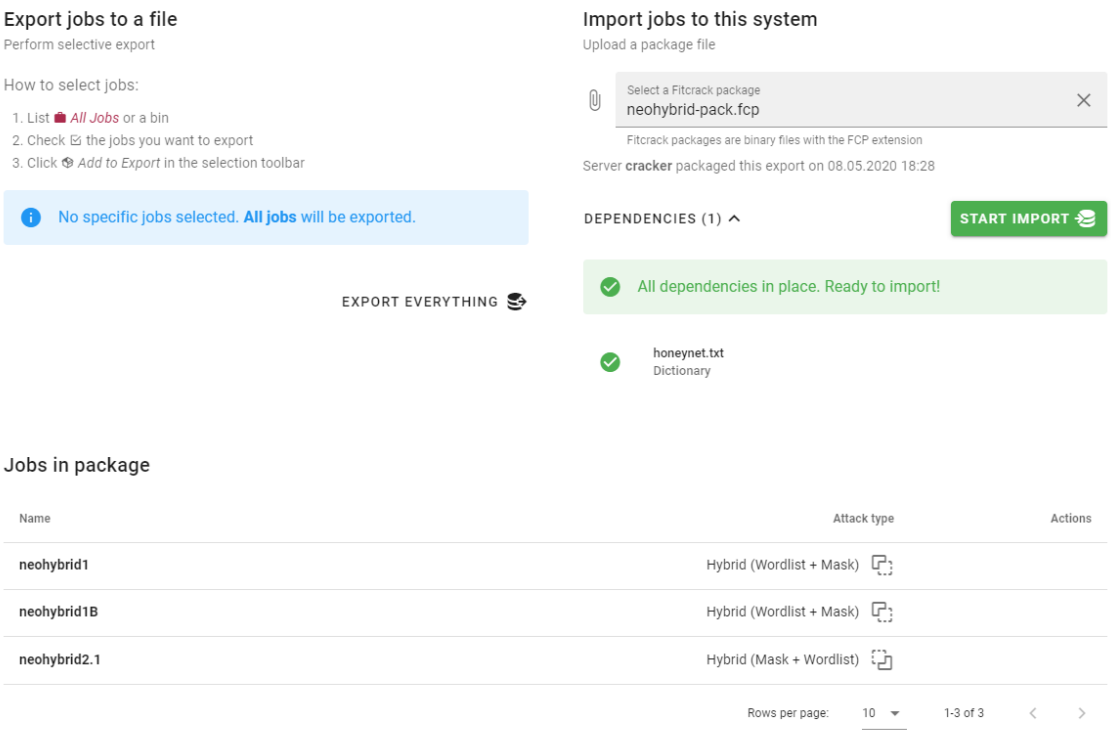
V rámci rozšiřování možností systému a úpravy aplikace jsem narazil i na další, menší oblasti potenciálně vhodné k vylepšení či přidání. Tyto nemají takový rozsah jako výše zmiňované, ale je třeba je zmínit a popsat.

4.12.1 Předvolby systému

Stránka předvoleb umožní přímo z administrační aplikace měnit globální nastavení systému. To je aktuálně možné měnit pouze změnou hodnot přímo v databázi. Navíc zde bude možné umístit i nastavení samotné aplikace, která by jinak nebylo kam zařadit. V obou částech pak bude možnost přidávat další volby v případě, že se systém rozšíří o další funkce či možnosti. Stránka je zachycena na obrázku 4.16.

4.12.2 Dekódování znakových sad

Aktuálně stránka s náhledem znakové sady zobrazuje ve většině případů nesprávné symboly. Problém je v tom, že soubor sady je pouze výčet znaků v určitém kódování a nelze automaticky rozeznat, ve kterém. Proto toto rozhodnutí padne do rukou uživatele, který si bude moci vybrat ze seznamu nebo uvést vlastní kódování, podle kterého se pak zobrazí



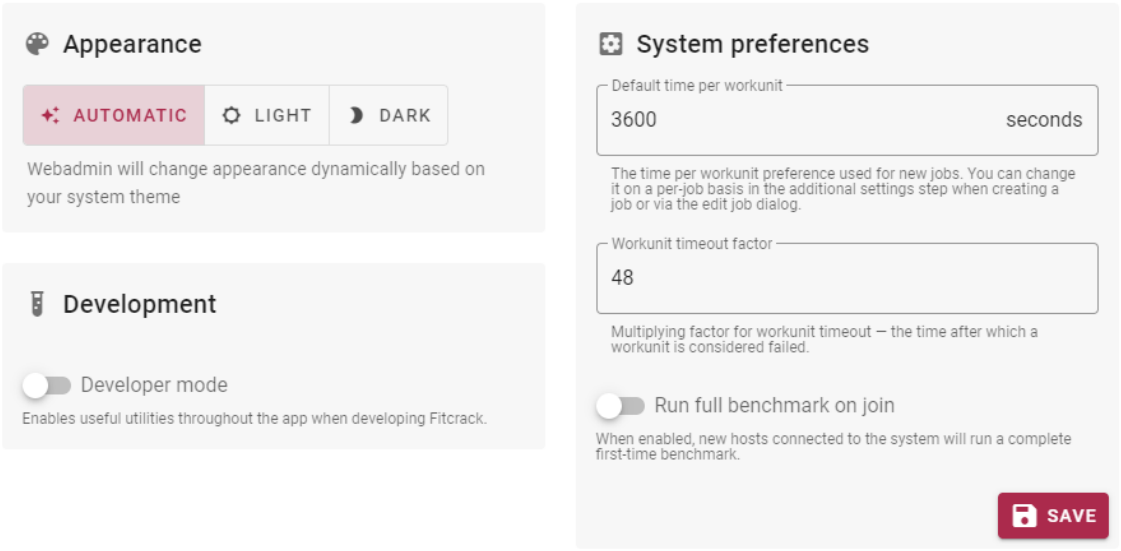
Obrázek 4.15: Stránka pro přenos dat před importováním

příslušné symboly. Aplikace se zároveň pokusí najít známý název kódování v názvu souboru a případně toto předvolit.

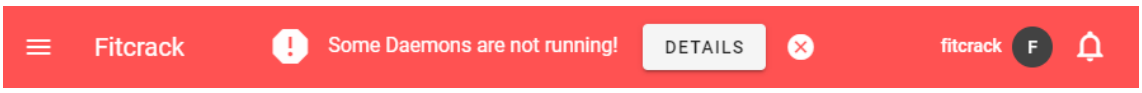
4.12.3 Sledování stavu služeb na pozadí

Pokud selže jedna nebo více služeb podporujících běh systému na serveru, může se Fitcrack chovat nepředvídatelně nebo vůbec nefungovat. V aplikaci na to přitom uživatel není nijak upozorněn. Pokud nenavštíví stránku sledování stavu serveru, nemusí se to vůbec dozvědět.

Nově se aplikace bude dotazovat na stav pravidelně a bez ohledu na aktuálně zobrazovanou stránku. V případě, že některé služby neběží, upozorní uživatele výrazné oznámení v horním panelu aplikace, jak znázorňuje obrázek 4.17. Součástí hlášení pak bude seznam postižených služeb a krátký popis, jak problém řešit.



Obrázek 4.16: Nastavení aplikace a předvolby systému Fitcrack



Obrázek 4.17: Upozornění na porouchané služby

Kapitola 5

Implementace

Každé rozšíření administračního systému z kapitoly 4 se dotýká různých jeho součástí. Klientskou aplikaci ovlivňují všechna, ovšem ne všechna také vyžadují změny na API serveru nebo v připojené databázi. Protože každá součást systému funguje odděleně a je implementována jinou technologií, uvedu u každého rozšíření implementaci změn a nových funkcí v relevantních částech zvlášť.

Verzování projektu

Jako základ pro veškerou implementaci v této práci je použita aktuální vývojová větev projektu ve službě Github[1]. Tato větev zahrnuje i mé změny klientské aplikace, které jsem prováděl mimo zadání, ovšem ty příliš nesouvisí s touto prací. Pro implementaci funkcionality z této práce jsem ze zmíněné větve oddělil vlastní větev *future*, do které ovšem byly často přidávány změny i z vývojové větve projektu. Toto jsem prováděl z důvodu jasnějšího a snadnějšího vývoje mých rozšíření v prostředí, kde nakonec budou fungovat. Mé úpravy a rozšíření zasahují do následujících adresářů:

- `/webadmin/fitcrackFE` – Webová aplikace administrace
- `/webadmin/fitcrackAPI` – Serverová část administrace
- `/server/sql` – Skripty SQL pro založení databáze
- `/server/src` – Zdrojové kódy generátoru a asimilátoru
- `/docs` – Uživatelský manuál pro web

[1] https://github.com/nesfit/fitcrack/tree/dev

5.1 Šablony úloh

Implementace šablon, ač z principu poměrně jednoduchá, vyžadovala radikální změnu celého formuláře zadávání úlohy v klientské aplikaci (3.3.1). Dosavadní implementace formuláře totiž nebyla navržena s úmyslem jej vyplňovat jinak, než vstupem přímo od uživatele.

Úpravy databáze

V rámci databáze bylo třeba přidat novou tabulku, *fc_template*, kam se budou šablony ukládat. Šablony se posílají po síti jako serializovaný objekt ve formátu JSON. Protože jediné operace se šablonami je jejich ukládání a nahrávání jako celku, rozhodl jsem se objekt nerozkládat do samostatných sloupců v tabulce. Namísto toho se celá šablona uloží v serializované textové podobě do sloupce typu longtext. Původně jsem použil přímo sloupec typu JSON, který umožňuje v případě potřeby provádět SQL dotazy i v rámci samotného dokumentu, ovšem pro potřeby šablon byl využit pouze jako úložiště. Změna na longtext byla provedena z důvodu kompatibility se staršími verzemi MySQL a Maria DB. Výsledné schéma tabulky navíc obsahuje sloupce s názvem a datem vytvoření, které slouží k výpisu uložených šablon. Schéma je znázorněno v tabulce 5.1.

fc_template	
id	int(11)
name	varchar(64)
created	timestamp
template	longtext

Tabulka 5.1: Schéma tabulky šablon

Úpravy serverové části

Na serveru bylo potřeba zprovoznit již existující koncový bod určený pro šablony. Za tímto účelem jsem nejprve vytvořil nový databázový model (v SQLAlchemy se implementuje jako odvozená třída) reprezentující novou tabulku šablon. Třídu popisuje výpis 5.1.

```
class FcTemplate(Base):
    __tablename__ = 'fc_template'

    id = Column(Integer, primary_key=True)
    name = Column(String(64), nullable=False)
    created = Column(DateTime, nullable=False, server_default=text("\
        CURRENT_TIMESTAMP"))
    template = Column(JSON, nullable=False)
```

Výpis 5.1: Model šablony

Jednotlivé sloupečky tabulky zde popisují instance třídy Column, jejíchž pojmenované argumenty odpovídají parametrům sloupečků v MySQL, jako je značení primárního klíče

nebo povolení nulové hodnoty. Další práce s modelem pak probíhá v jednotlivých metodách definujících v API akce dostupné na koncových bodech. V rámci šablon to jsou:

- `GET /template` – vrátí seznam uložených šablon,
- `PUT /template` – uloží do tabulky šablonu přijatou jako JSON,
- `GET /template/<id>` – vrátí obsah šablony podle id.

Za účelem dotazování existuje na modelu vlastnost `query`. Vytvoření nového záznamu se pak provádí vytvořením instance třídy modelu a přidáním přes objekt `db.session` a jeho metody `add` a `commit` pro potvrzení transakce.

Úpravy klientské aplikace

Ve Vue se data mezi jednotlivými komponenty předávají potomkům pomocí atributů (props) a nazpět rodičům pomocí událostí (events). Pro složitější kompozice tato hierarchie často znamená markantní nárůst složitosti kódu. V případě formuláře pro zadávání úloh v systému Fitcrack navíc jednotlivé komponenty často předávaly svým rodičům data získaná zpracováním několika údajů ze svých potomků do jiné podoby.

Pokud bychom chtěli v takové architektuře do elementárních komponent hluboko v této hierarchii zadávat data ze šablony, vyžadovalo by to na každé úrovni data opět „rozebrat“ do jiné podoby. Pro tak složité zobrazení, jako je tento formulář, je něco takového nepřijatelné.

Jako možné řešení se nabídlo centrální úložiště stavu s jednosměrným tokem dat, v případě Vue knihovna Vuex[2]. Více komponent pak může jako zdroj dat využít sdílené místo, kde jsou dovolené změny (mutace) předem definované.

Pro formulář jsem se rozhodl úplně nahradit předávání dat právě sdíleným úložištěm a do něj také zapouzdřit veškerou logiku vyhodnocování platnosti zadaných dat a sestavování výsledného objektu nastavení, který se zasílá na server.

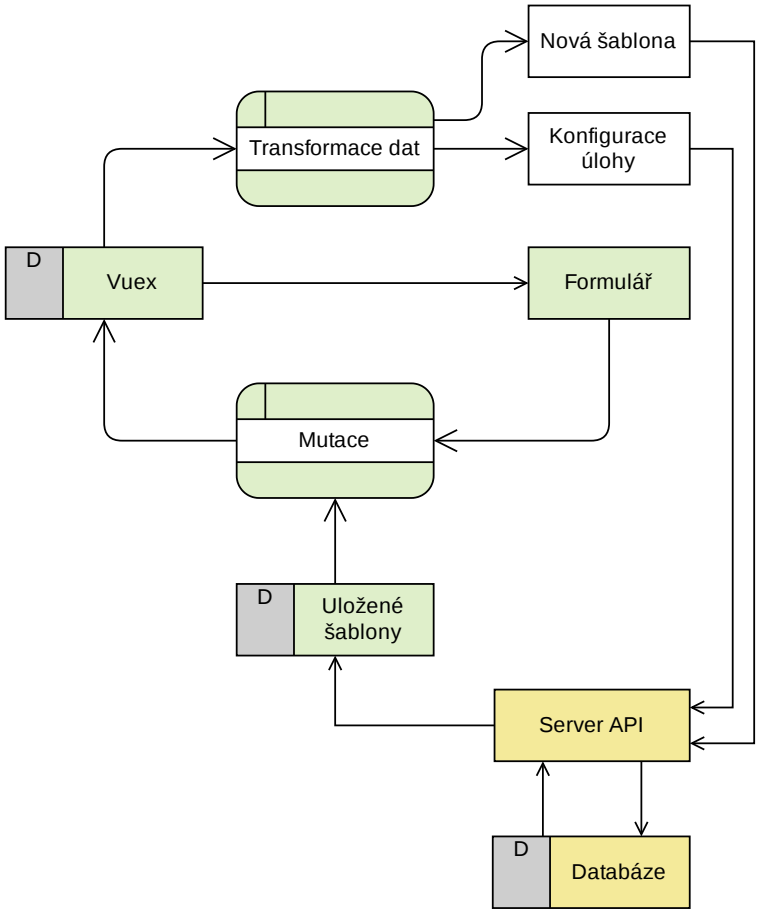
Díky této změně je možné celý stav formuláře ovlivnit i „zvenčí“ a tím pádem aplikovat právě šablony, které jsou pouze serializované obrazy stavu z modulu formuláře v úložišti Vuex. Navíc, jelikož je úložiště platné v rámci celé aplikace, zůstává v něm aktuální stav konfigurace uložen i při opuštění stránky formuláře. Při návratu se komponenty obnoví do předchozího stavu, protože svá data čerpají právě odtud. Diagram na obrázku 5.1 znázorňuje tok dat v takto koncipovaném systému.

5.2 Skupiny úloh

Skupiny měly podle původního návrhu plnit dvojí účel. Kromě třídění úloh do pojmenovaných pohledů měly zahrnovat i funkcionalitu dávek. Postupem času se ale ukázalo, že kombinace obojího je velmi komplikovaná, protože některé implementační detaily si mezi těmito dvěma funkcemi odporují. Proto jsou dávky nakonec úplně samostatnou částí jak v návrhu, tak implementaci, popsané v části 5.3. Skupiny tedy slouží pouze k roztřídění úloh a jsou integrovány přímo do jádra administrační aplikace.

Hlavním důvodem pro rozdělení je odlišná relace úloh ke skupinám a dávkám. Zatímco v případě dávek může každá úloha náležet pouze do jedné fronty, skupiny by měly umožnit uživateli zařadit stejnou úlohu do vícero skupin.

[2] https://vuex.vuejs.org/



Obrázek 5.1: Tok dat mezi formulářem, jeho úložištěm stavu a API

Úpravy databáze

Pro ukládání samotných skupin, které se pak zobrazují v postranním panelu, slouží tabulka *fc_bin*. Její strukturu popisuje tabulka 5.2. V této tabulce se ukládá název skupiny a její umístění v seznamu, které lze v aplikaci měnit. Pro přiřazení úloh do skupin ve vztahu M:N pak slouží spojovací tabulka *fc_bin_job*, která každým záznamem spojuje ID konkrétní úlohy s ID konkrétní skupiny. Její schéma pak znázorňuje tabulka 5.3.

fc_bin	
id	int(11)
name	varchar(50)
position	int(11)

Tabulka 5.2: Schéma tabulky skupin

Dále je potřeba zajistit funkcionalitu přesouvání skupin v seznamu na libovolné místo. Skupiny jsou v systému společné pro všechny uživatele, včetně pozice, a měnit je mohou pouze správci. Proto jsem potřeboval údaj o pozici také zaznamenat do databáze. Zde se nabízelo několik způsobů provedení.

Možnost, kterou jsem nakonec zvolil, je ukládání skutečné pozice ve formě pořadového čísla, které je při přesunu položky nutné upravit i u ostatních záznamů. I když je tento

fc_bin_job	
id	int(11)
job_id	bigint(20)
bin_id	int(11)

Tabulka 5.3: Schéma spojovací tabulky skupin a úloh

způsob náročnější na zásahy do databáze, nabízí několik výhod. Jednak je stále poměrně jednoduchý na implementaci. Dále je pak, za předpokladu, že se čísla upravují správně, spolehlivý a to bez ohledu na rozsah úpravy a četnost položek v seznamu.

Alternativní způsob, jako je například ukládání pozic s rozestupem a přesouvaní do vzniklých „mezer“, umožňuje méně náročný přesun. Po určitém počtu přesunů položek, který závisí na velikosti rozestupu, se ovšem již nenajde místo pro vložení přesouvané položky. V té chvíli je potřeba provést přeskupení celého seznamu a vytvoření nových rozestupů, jinak by operace nemohla být dokončena. To znamená, že tento způsob pouze odsouvá režii spojenou se změnou pořadí v celém seznamu na později. To také v důsledku znamená celkově složitější logiku.
Při práci se skupinami je potřeba brát ohled na pořadí v těchto případech:

- vytvoření nové skupiny (přiřazení pozice na konci seznamu),
- přesun skupiny (změna pořadí všech položek mezi původní a novou pozicí),
- smazání skupiny (přesun položek o pozici níže pro zaplnění volného místa).

Jelikož se často jedná o transakce provádějící několik dotazů za sebou a je potřeba je provádět spolehlivě a pokud možno rychle, rozhodl jsem se je implementovat jako uložené podprogramy v systému řízení báze dat (dále už jen DBMS[3]). V případě uložených procedur pak tyto volá klient SQLAlchemy ze serverové části.

Pro přiřazování pozice nově vytvořené skupině slouží spoušť (trigger) z výpisu 5.2, která novému záznamu přidá pozici za nejvyšší aktuálně existující nebo, pokud se jedná o první záznam, pozici první (0).

```
declare pos int;
set pos = (select max(position) from fc_bin);
if pos is null then
  set NEW.position = 0;
else
  set NEW.position = pos + 1;
end if;
```

Výpis 5.2: Spoušť obsluhující přidání skupiny

Pro změnu pozice ostatních položek při přesunu nebo smazání záznamu skupiny již spouště využít nelze. Důvodem je jedno z omezení MySQL na spouštích, které jim zakazuje zasahovat do tabulky, která je již používaná [10]. To znamená, že v dané tabulce, ze které pochází záznam, jenž vyvolal provádění spouště, lze měnit pouze tento záznam. Aby tedy bylo možné v rámci DBMS provádět požadované změny, je třeba využít uložených procedur.

Obě procedury, pro přesun i mazání, mají stejnou základní strukturu. Kromě parametrů, deklarace proměnných a samotného těla uvnitř transakce zahrnují omezení vykonávání na

[3]Database Management System

oprávnění volajícího pomocí `sql security invoker` a obsluhu výjimek provádějící vrácení změn. Postup přesunu položky v seznamu popisuje algoritmus 1. V případě mazání skupiny se pak jen sníží pozice všech položek nad smazanou, aby nevznikla mezera.

Algoritmus 1: Přesun položky v seznamu skupin

```
pokud cílová pozice > aktuální pozice
    pro všechny položky mezi aktuální a cílovou pozicí proveď
        pozice položky −= 1
jinak
    pro všechny položky mezi cílovou a aktuální pozicí proveď
        pozice položky += 1
pozice přesouvané položky = cílová pozice
```

Úpravy serverové části

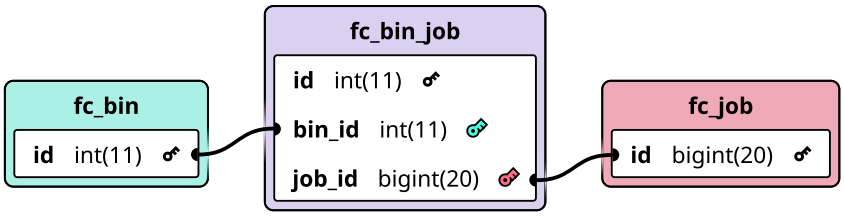
Skupiny jsou v rámci serverové části úplně nový pojem, takže pro ně neexistoval žádný dříve implementovaný základ, jako tomu bylo u šablon (viz část 5.1). Pro API postavené za pomocí frameworku Flask nad SQLAlchemy zavádění nového podsystému zahrnuje:

- vytvoření modelů pro tabulky databáze,
- zavedení relací mezi modely,
- vytvoření jmenného prostoru pro koncové body,
- definici koncových bodů a jejich logiky,
- definici vstupních rozhraní a čtení argumentů,
- vytvoření modelů odpovědí.

Modely pro tabulky skupin (*FcBin* a *bin_job_junction*) jsou vytvořeny obdobně, jako u šablon, jak ukazuje výpis 5.1. Nově je ale potřeba také definovat relaci mezi tabulkami, aby objektový model mohl fungovat v rámci jejich M:N vztahu. Relaci tabulek na obrázku 5.2 popisuje v objektovém modelu SQLAlchemy konstrukce z výpisu 5.3. Parametr `secondary` specifikuje spojovací tabulku a `backref` umožňuje zároveň přidat i druhou stranu relace do druhé tabulky. V tomto případě bude instance modelu *FcBin* zahrnovat v *jobs* všechny jí přiřazené úlohy. Zároveň pak bude v modelu *FcJob* dostupné pole *bins*, kde budou zahrnuty všechny skupiny, do kterých úloha patří.

```
jobs = relationship('FcJob',
                    secondary=bin_job_junction,
                    backref="bins",
                    lazy='dynamic',
                    passive_deletes=True)
```

Výpis 5.3: Popis vztahu tabulky skupin k tabulce úloh v modelu SQLAlchemy



Obrázek 5.2: Schéma vztahu M:N mezi tabulkou skupin a tabulkou úloh

Pro vystavení koncových bodů API slouží jmenný prostor. Ten shromažďuje související koncové body pod stejnou základní cestou. Zároveň pro přehlednost tvoří oddělenou sekci v generované dokumentaci. Pomocí `api.namespace('bins')` jsem tak vytvořil jmenný prostor pro skupiny, který zahrnuje všechny koncové body pro práci s nimi. Jsou to:

- **GET /bins** – vrátí seznam existujících skupin
- **POST /bins** – vytvoří novou skupinu s určeným jménem
- **PATCH /bins/<id>** – upraví skupinu s daným *id* (přejmenování apod.)
- **DELETE /bins/<id>** – odstraní skupinu s daným *id*
- **PATCH /bins/<id>/move** – přesune skupinu s daným *id* na určené místo
- **PUT /bins/<id>/assign** – upraví přiřazení úloh ke skupině s daným *id*

Koncové body pro přesun a mazání skupin nepoužívají objektový model, ale přímým spojením s DBMS volají výše zmiňované procedury. K tomu slouží v SQLAlchemy metoda `callproc`. Koncový bod měnící přiřazení úloh přijímá jako argument objekt, který obsahuje pole ID úloh k přidání a pole ID úloh k odstranění ze skupiny. Lze tak jedním voláním provést přidání i odebrání úloh.

Úpravy klientské aplikace

V aplikaci je seznam skupin začleněný jako komponenta do navigačního panelu, jak je popsáno v částech 4.3 a 4.4. Skládá se z hlavičky se skrytým vstupním polem pro vytváření skupin a samotného seznamu. Možnost vytvářet skupiny a měnit jejich pozici je podmíněna oprávněním uživatele.

Při vytváření se automaticky přesouvá zaměření na stránce na vstupní pole a zpět na tlačítko vytvoření skupiny pro opakované použití. Vstupní pole také reaguje na klávesové příkazy. Klávesa enter potvrdí zadaný název a vytvoří skupinu, zatímco escape zruší vytváření.

Veškeré akce spojené se skupinami a stav samotného seznamu jsou opět uloženy v modulu úložiště Vuex (viz část 5.1). Akce[4] jsou asynchronní funkce, které zajišťují i zasílání požadavků REST na server. Důvodem pro využití modulu Vuex je nutnost komunikovat s touto komponentou i z výpisu úloh (a případně dalších míst) kvůli přiřazování úloh do skupin. Veškerá logika spojená se skupinami je tak na jednom místě.

[4] https://vuex.vuejs.org/guide/actions.html

Pro přesun skupin v seznamu pomocí gesta tažení je využita komponenta VueDraggable[5]. Ta umožňuje vytvořit seznamy s pohyblivými položkami při zapouzdření složitější logiky. Přesunutí a „upuštění“ položky v seznamu vyvolá událost, která oznamuje, která položka ji vyvolala, a původní a nový index v seznamu. Tyto informace se postupně předají až výše zmíněné proceduře, která provede úpravy v databázi a aplikace dostane odpověď s výsledkem. Během zpracování je přesun zakázán, aby nedošlo ke konfliktům, i když za normálních podmínek je přesun vyřízen v podstatě okamžitě.

5.3 Dávkové spouštění

Dávkové spouštění úloh, které mělo původně být součástí skupin úloh, řeší podobné, ovšem ne zcela stejné, problémy jako skupiny. Aby dávky mohly běžet v sekvenci bez zásahu uživatele, bylo potřeba upravit i části logiky generátoru a asimilátoru tak, aby k nastavení stavu úloh používaly novou proceduru zajišťující i spouštění případných následníků. Proceduru blíže popisuje následující část.

Úpravy databáze

Pro systém dávek jsem potřeboval navrhnout způsob, jak ukládat pořadí úloh a jejich příslušnost dané dávce tak, aby mohla automaticky být spuštěná následující úloha po dokončení té předcházející. Vázaný seznam pomocí sloupečku odkazujícího na ID následující úlohy se jevil jako vhodný kandidát, ale pro implementaci v relační databázi není příliš vhodný. Pokud bychom totiž chtěli získat celý seznam pro výpis ve správném pořadí, museli bychom namísto nativního seřazení databází procházet odkazy v seznamu.

Namísto toho jsem využil kombinace pořadových čísel použitých v seznamu skupin (viz část 5.2) a dotazování v rámci relace 1:N mezi dávkou a úlohami. Každá úloha tak má sloupeček pořadí, který je vyplněn vždy po přiřazení k dávce. Pokud úloha není součástí dávky, pořadí ztrácí význam a ignoruje se. V rámci dávky tak lze jednoduchým dotazem získat záznam následující úlohy, když se současná dokončuje. Dotaz se omezí jen na úlohy ve stejné dávce, jako je současná úloha a dohledá se záznam s nejbližším vyšším pořadovým číslem. Pokud žádný záznam není nalezen, pak dávka dokončila zpracování celé fronty.

Samotné dávky se ukládají do tabulky, jejíž strukturu znázorňuje tabulka 5.4. Sloupeček *creator_id* se odkazuje do tabulky uživatelů a slouží jako identifikátor tvůrce dávky. Úlohy se k dávce přiřazují cizím klíčem pomocí sloupečku *batch_id* v jejich záznamech.

fc_batch	
id	int(11)
name	varchar(50)
creator_id	int(11)

Tabulka 5.4: Schéma tabulky dávek

K automatickému spuštění následující úlohy slouží nová procedura, kterou volají asimilátor a generátor namísto přímého nastavení stavu běžící úlohy. Procedura po nastavení stavu aktuální úlohy provede výše popsaný dotaz na následníka a případně jej spustí. K tomu dochází zpravidla při přechodu do stavu dokončování, protože v tu chvíli již generátor

[5] https://github.com/SortableJS/Vue.Draggable

nevytváří pro uzly nové pracovní jednotky. Kód těla procedury je ve výpisu 5.4. Aby nedocházelo k nechtěnému spouštění dalších úloh po zastavení, kontroluje se při přechodu do těchto stavů, zda již další úloha neběží. Takto je možné zachovat sekvenční povahu dávek, ale také spouštět následníky i v případě, že úloha z jakéhokoliv důvodu přeskočila stav dokončování. Protože procedura obsluhuje veškeré změny stavu pro běžící úlohy, je třeba brát ohled na uživatelem vyžádané zastavení úlohy. Pokud je tedy nový stav 0, spuštění následníka se také neuskuteční.

```
-- nastaveni noveho stavu ulohy
update fc_job set status = new_status
where id = job_id and status >= 10 limit 1;

-- vyhledani informaci o davce
select batch_id, queue_position
into current_batch_id, current_queue_position
from fc_job where id = job_id;

-- dohledani naslednika
select id into succ_id
from fc_job where batch_id = current_batch_id
and queue_position > current_queue_position
and status = 0 -- ready
order by queue_position asc
limit 1;

-- zjisteni poctu bezicich uloh v davce, pokud se aktualni zastavuje
set running_count = 0;
if new_status between 1 and 9 then
  select count(id) into running_count
        from fc_job where batch_id = current_batch_id
        and status >= 10;
end if;

-- start naslednika, pokud jeste nebezi
if succ_id is not null and running_count = 0 and new_status <> 0 then
  update fc_job set status = 10 where id = succ_id;
end if;
```

Výpis 5.4: Tělo procedury nastavující stav úlohy a spouštějící následující úlohu dávky

Úpravy serverové části

Stejně jako u skupin jsem na straně serveru vytvořil nový jmenný prostor API pro práci s dávkami. Jeho koncové body jsou následující:

- **GET /batches** – Vrací po stránkách seznam dávek
- **PUT /batches** – Vytvoří nebo upraví dávku
- **GET /batches/<id>** – Vrátí dávku podle ID, včetně úloh
- **DELETE /batches/<id>** – Smaže dávku podle ID
- **POST /batches/<id>/run** – Spustí první připravenou úlohu ve frontě
- **POST /batches/<id>/interrupt** – Zastaví právě běžící úlohy

Stav dávky je odvozen z úloh ve frontě. Pokud nějaká běží, dávka se také považuje za probíhající. Pokud žádná neběží, dávka se považuje za přerušenou, případně skončenou, pokud jsou všechny úlohy skončené. Stejně pak i server dávky ovládá – spuštění dávky znamená spuštění první připravené úlohy a přerušení zastaví běžící úlohu.

Vytváření i úpravu dávek zajišťuje stejná metoda, která přijímá název a seznam ID úloh. Pokud dávka již existuje, pouze se přepíše. Pořadová čísla úloh se nastavují v závislosti na pořadí ID v seznamu. Smazání dávky znamená odstranění záznamu z databáze. Při tom se díky zadanému omezení smaže přiřazení z příslušných úloh.

Pro dávky jsem navíc rozšířil koncové body pro grafy podílu výpočetních uzlů a rozdělení pracovních jednotek. Nově jim lze zadat více než jednu úlohu a získat tak grafy se statistikou pro celou dávku. Změny v logice jsou minimální. Podíl výpočetních uzlů počítá součet z pracovních jednotek, takže pro více úloh je postup stejný. V případě rozdělení pracovních jednotek v čase je každá úloha v grafu reprezentována jako samostatná množina dat odlišnou barvou.

Úpravy klientské aplikace

V aplikaci je pro vytváření dávek určen nový dialog ve výpisu úloh (viz obrázek 4.10), který využívá stejnou komponentu k řazení úloh, jako seznam skupin, tedy VueDraggable. Ta mění pořadí položek v poli, které je v dialogu reprezentováno, takže při zasílání na server je již seřazené tak, jak jej uživatel zanechal.

Náhled rozložení náročnosti úloh využívá stejnou techniku, jako zobrazení pracovních jednotek v detailu úlohy. Celá vizualizace je umístěna v prvku s rozložením flexbox. Každou úlohu reprezentuje prvek, jehož hodnota *flex-grow* je rovna počtu hesel, která se budou zkoušet. Tuto hodnotu lze použít přímo, protože *flex-grow* udává poměr „roztažení“ prvku ve vztahu k ostatním prvkům v témže kontejneru [2].

Výpis existujících dávek je obsažen v tabulce podobné výpisu úloh na dedikované stránce *Batches*. Detail dávky pak zobrazuje úlohy ve frontě jako seznam a umožňuje jejich pořadí dodatečně změnit. V detailu se také zobrazuje celkový průběh, který je vypočítaný z průběhu všech úloh. Ve spodní části se nachází grafy se statistikami, které jsou popsány výše v části o úpravách serverové části.

5.4 Detail úlohy

Při přestavbě zobrazení detailu úlohy jsem se chtěl zaměřit nejen na přehlednost po vizuální stránce, ale i v kódu. Tam, kde to nekomplikovalo předávání dat, jsem oddělil funkční celky do samostatných komponent.

Hlavní problém velkých monolitických komponent ve Vue je rychle rostoucí nepřehlednost kódu. S rostoucí velikostí šablony dokumentu a počtem datových položek, metod a dalších součástí, roste i nutnost přesouvat se na potenciálně velmi vzdálená místa v kódu komponenty. Důvodem je organizace kódu podle kategorie. Například všechny definované metody se nachází v objektu *methods* ve skriptu komponenty, ale kód používající tyto metody je rozmístěn v šabloně. Tento problém se snaží řešit kompoziční API[6] zaváděné ve verzi 3 frameworku Vue, která bohužel v čase vývoje projektu neměla stabilní vydání.

Samotná stránka pak vychází z návrhu (obrázek 4.12) a z velké části mu odpovídá. Ukazatel historie stavu úlohy jsem nakonec postavil do vodorovného rozložení k získání místa pro ostatní detaily úlohy a výpis hešů. Historie stavu se navíc zobrazuje jen od posledního připraveného stavu (*ready state*) a zbytek je skrytý v dialogu. Tímto se skryje historie předcházející vynucenému obnovení úlohy (*purge*).

Postranní panel s hlavními informacemi je generován dynamicky podle dostupných informací a zároveň slouží jako kontejner pro komponentu editace. Většina ostatních součástí si zachovala původní podobu a jsou pouze rozmístěny v novém rozložení z návrhu. Výsledná podoba je na obrázku 5.3.

Sekce zobrazující informace o podílejících se uzlech a sekce s grafy statistik jsou rozděleny do sloupců s pevně daným poměrem šířky. Důvodem je přítomnost právě grafů, které se musí překreslovat dynamicky v reakci na změnu velikosti jejich rodičovského prvku. V kontextu rozložení flexbox prvky grafů reagují správně na zvětšení jejich kontejneru, ale při zmenšování již nerespektují poměry se sousedními prvky. Tím dojde k tomu, že jeden prvek grafu „vytlačí“ druhý na další řádek a rozložení do sloupců se nenávratně rozpadne. Toto chování jsem vypozoroval u několika různých knihoven vykreslujících grafy, ať už byly založeny na kreslení SVG[7] nebo využívaly Canvas[8], protože v obou případech je potřeba při každé změně celý obrázek přepočítat.

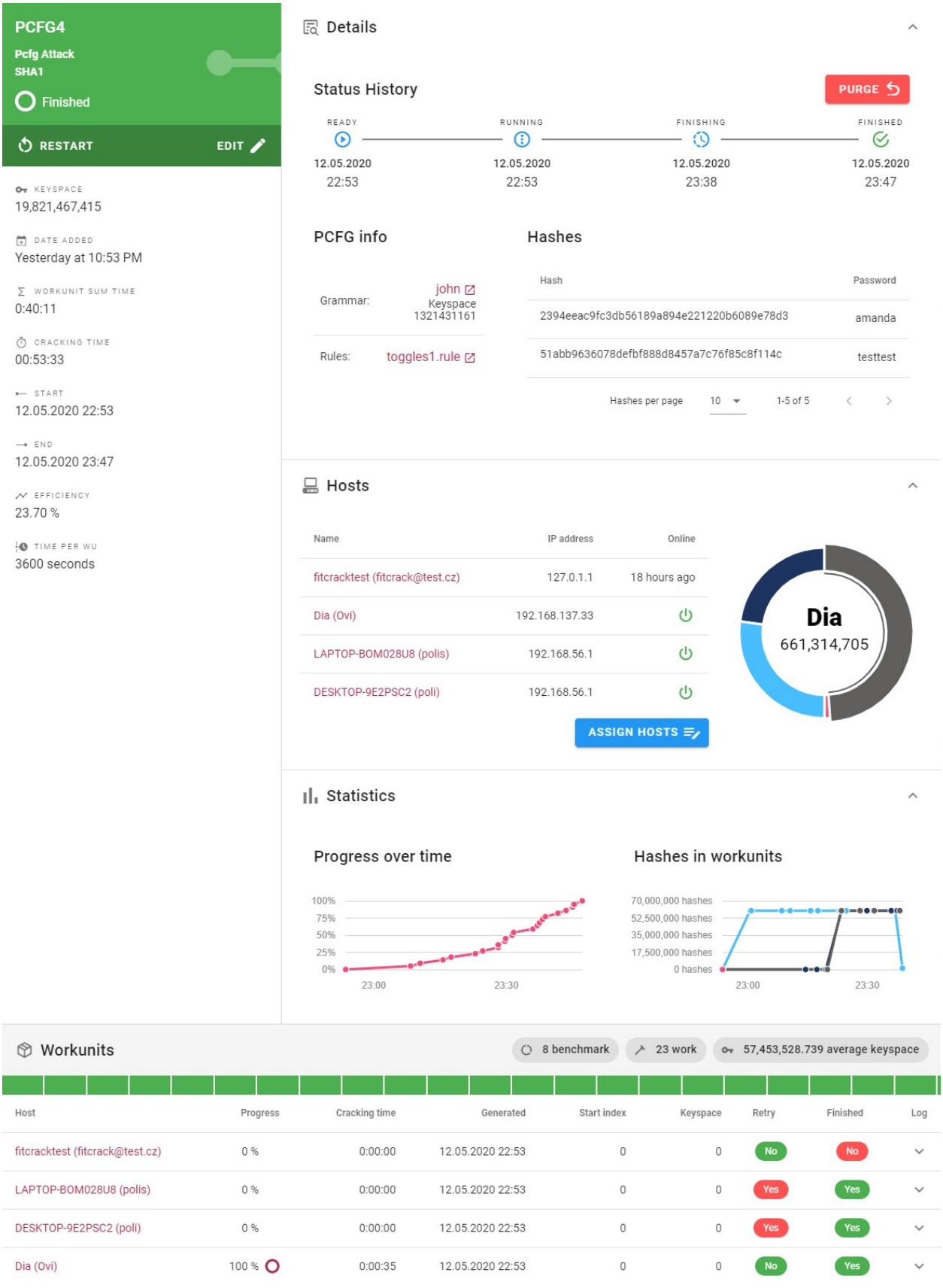
5.5 Prohlížeč PCFG

K procházení obsahu gramatik PCFG musí být uživateli prezentována celá adresářová struktura gramatiky, protože sestává z více souborů v různých kategoriích. Samotný obsah souborů je pak načítán dynamicky, podle výběru uživatele.

Úpravy serverové části

Pro účely zobrazení adresářového stromu jsem se rozhodl vždy posílat celou strukturu ve formě teoreticky libovolně hluboko vnořeného objektu. Každý adresář je objekt obsahující mimo jiné pole dalších podobných objektů. Záznamy souborů toto pole neobsahují. V rámci gramatik obvykle hloubka zanoření nepřesahuje jednu úroveň, ale systém je na hlubší zanoření podadresářů připraven i tak.

[6] https://composition-api.vuejs.org
[7] https://www.w3.org/TR/SVG11/
[8] https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API



Obrázek 5.3: Výsledná podoba detailu úlohy (dokončený útok PCFG)

Rekurzivní průchod adresáři provádí funkce `walk` z knihovny *os* jazyka Python. Ta ovšem vrací plochý seznam záznamů s cestami k položkám. Moje funkce, vracející zanořený slovník, pak v cyklu pro každý tento záznam slovník prochází od kořenu udávanou cestou a do cíle vloží daný obsah.

Pro výpis obsahu souboru pak slouží obdobná funkce, jako u ostatních prostředků uložených v souborech. Posílá obsah jako textová data po přečtení ze souboru na serveru. Novou funkcionalitu vystavuje jmenný prostor *PCFG* pomocí dvou nových koncových bodů:

- **GET /pcfg/<id>/tree** – Vrátí rekurzivní objekt reprezentující strom adresářů
- **GET /pcfg/<id>/file** – Vrátí obsah vybraného souboru z gramatiky

Úpravy klientské aplikace

V aplikaci jsou z názvů gramatik v tabulce odkazy vedoucí na novou stránku prohlížeče. Stránka po navigaci požádá API o strom adresářů. Ten je pak reprezentován komponentou *Treeview*, která ve výchozím nastavení přijímá právě takový rekurzivní objekt, jaký je vytvářen podle popisu výše. Po vybrání libovolného souboru ze stromu se vyžádá obsah souboru podle konkrétní cesty a zobrazí se v sousední kartě vedle stromu.

5.6 Systém oprávnění

Pro systém oprávnění již existovaly v systému základy. Většina implementace ale doposud chyběla. Navíc jsem celý koncept rozšířil o pojem vlastníků, kteří mají možnost přidělovat ostatním oprávnění k manipulaci jejich úloh.

Úpravy databáze

V databázi již existovaly tabulky rolí s oprávněními vázané na uživatele, i když z většiny nevyužité. Tabulka pro jednotlivá oprávnění ke konkrétním úlohám nebyla využita vůbec, protože mělo od tohoto konceptu být upuštěno. Tato tabulka obsahuje vazby na úlohu a na uživatele a doplňuje jednobitové značky pro přidělená oprávnění. K těmto jsem ještě přidal další značící vlastníka úlohy. Tato značka má přednost před ostatními a vlastník má vždy všechna oprávnění na svojí úlohu.

Úpravy serverové části

Většina úprav serverového kódu se týkala zabezpečení a omezení různých koncových bodů na základě oprávnění přihlášeného uživatele. Ten je udržován pro každé sezení jako proměnná `current_user` systémem přihlášení *flask-login*. Proměnná uchovává přímo instanci databázového modelu, takže s ním lze přímo pracovat a číst kýžené údaje jako přidělené role. V případě vypisování úloh lze jeho ID využít pro vyhledání úloh na základě jemu přidělených práv ke konkrétním úlohám.

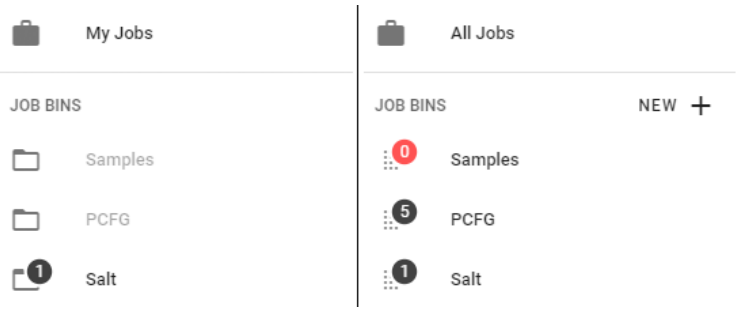
Pro rozšíření o funkcionalitu vlastníků jsem do modelu úlohy přidal vlastnost *permissions*. Ta obsahuje slovník se značkami z tabulky oprávnění pro aktuálního uživatele. Díky tomu není potřeba provádět zvláštní dotaz na oprávnění, pokud jde o práva přihlášeného uživatele. Aby ale mohl vlastník úlohy tyto práva měnit, musel jsem zavést pro úlohy koncový bod `/job/<id>/permissions` s metodami **GET** (vrací seznam uživatelů a jejich práv) a **PUT** (přidává nebo mění uživatele v seznamu). Parametry jednotlivých práv jsou

v metodě **PUT** volitelné. Pokud je přidáván nový uživatel, výchozí hodnota neuvedeného parametru je *zakázáno*, jinak zůstane nezměněná.

Úpravy klientské aplikace

Největší rozšíření v aplikaci představuje dialog pro přidělování oprávnění k úlohám (viz obrázek 4.14). Ten je dostupný při editaci úlohy tlačítkem *Permissions*. Dialog využívá výše popsaný koncový bod. Fungování je velice podobné přidělování rolí na stránce správy uživatelů. Kliknutím na zaškrtávací pole oprávnění u uživatele se vyšle požadavek na změnu jeho záznamu. Pro přidání nového uživatele k úloze slouží vyhledávací seznam, který čerpá data z bodu `/user/`.

Na stránce správy uživatelů jsem pak přidal možnost přiřazovat zbytek rolí, které doposud nebyly využité. Ostatní změny se týkají různých částí rozhraní, které jsou podmíněny rolemi uživatele. Pokud uživatel nemá oprávnění vytvářet, upravovat, či ovládat některou součást, daný ovládací prvek je skryt nebo zakázán. Příklad je na obrázku 5.4. Uživatel s právy administrátora může zobrazovat a měnit všechny úlohy v systému a tím pádem i skupiny. Naproti tomu uživatel bez těchto rolí může vidět pouze své úlohy a ty k nimž mu byla práva přidělena, což odráží i navigační panel. Tyto změny samozřejmě podporují zabezpečení zavedená na serveru, jinak by uživateli stačilo posílat požadavky přímo na API a jednoduše by je obešel.



Obrázek 5.4: Srovnání nabídky úloh uživatele se základními právy (vlevo) a administrátora (vpravo)

5.7 Export a import úloh

Podobně jako zavedení šablon úloh byl export a import dat něco, s čím původně architektura systému nepočítala. Namísto úplné změny vnitřního fungování systému jsem ale naopak potřeboval řešení, které by do něj nijak nezasáhlo. Zálohovací systém tak funguje svým způsobem mimo Fitcrack, ne nepodobně jako prosté zálohy obsahu databáze. Stále je však součástí logiky serverové části a dokáže provádět nutné kontroly a transformaci dat tak, aby se dala přenášet.

Úpravy serverové části

Návrh, jak data úloh transformovat a uložit v serializované podobě byl klíčový. V systému Fitcrack jsou totiž úlohy závislé na různých dalších prostředcích, které se navíc různí podle typu útoku a dalších voleb. Tyto prostředky pak mohou dosahovat velikostí, které jsou

nepraktické pro přidání do balíku zálohy. Navíc tyto prostředky, jako jsou například slovníky hesel nebo sbírky masek, uživatel do systému dříve sám přidával, takže je může případně sám do cílového systému doplnit a nebude nucen je znovu stahovat v exportovaném balíku.

Pro účely serializace jsem proto model úlohy rozdělil na pole, která se budou exportovat přímo, a na závislosti. Závislosti jsou typicky odkazy na záznamy v jiných tabulkách, které ukládají informaci o souboru prostředku umístěném na serveru. Protože mohou být sdílené mezi vícero úlohami, ukládají se do zálohy v seznamu zvlášť a jednotlivé úlohy se na ně odkazují indexem.

Při ukládání samotných závislostí nelze spoléhat na identifikátory z databáze, protože v cílovém systému mohou být v jiném uspořádání. Ukládá se proto hodnota z databáze, podle níž lze obvykle prostředek rozeznat, typicky název spolu s typem prostředku. Podle tohoto se pak v cílovém systému nástroj snaží prostředky dohledat. Celý postup ukládání jedné úlohy popisuje algoritmus 2.

Algoritmus 2: Ukládání úlohy při exportování

```
Vstup: seznam již uložených závislostí
úloha = {}                                              // prázdný objekt
pro každý sloupec z přímo exportovaných jako col proveď
    úloha[col] = zdroj[col]
pro každý typ závislosti jako dep proveď
    prostředky = [ ]                                    // prázdný seznam
    data = zdroj[dep]
    pokud data chybí nebo je seznam prázdný
        přeskoč iteraci smyčky
    pro každý prvek v seznamu data proveď
        pokud je prvek již uložen v seznamu závislostí
            index = index existující závislosti v seznam závislostí
        jinak
            přidej prvek jako závislost do seznamu závislostí
            index = poslední index v seznamu závislostí
        přidej index do seznamu prostředků
    úloha[dep] = prostředky
vrať objekt úlohy
```

Po uložení všech úloh je záloha reprezentována slovníkem jazyka Python, který obsahuje seznam úloh a seznam závislostí. Pro exportování se ještě musí serializovat, aby mohl být uložen do souboru. I když by se dal jednoduše využít formát JSON, rozhodl jsem se pro binární formát MessagePack[9]. Primárním důvodem byla menší výstupní velikost, ale také díky jeho binární formě nepřímo naznačuje uživateli, že by soubor neměl ručně upravovat.

Nahrávání souboru se zálohou do cílového systému předchází kontrola přítomnosti závislostí uvedených v balíku. O tu požádá klient a podle odpovědi případně informuje uživatele o nutnosti prostředky doplnit. Kontrola probíhá přečtením typu a hodnoty závislosti a dohledáním v databázi pomocí příslušného modelu.

Po nahrání souboru se odkazy na záznamy závislostí z databáze uloží v paměti a následně se pro každou úlohu v balíku vytvoří nová instance modelu úlohy. Do té se zkopírují přímo

[9] https://msgpack.org

uložené informace a připojí se k ní odkazy na závislosti. Tak vznikne v podstatě kopie úlohy z původního systému.

Systém pro exportování a importování dat obsluhují tyto nové koncové body ve jmenném prostoru *serverInfo*:

- **GET /serverInfo/transfer** – Pošle balík zálohy s vyžádanými úlohami
- **POST /serverInfo/transfer** – Zpracuje nahraný soubor zálohy
- **POST /serverInfo/transfer/validate** – Zkontroluje přítomnost prostředků

Úpravy klientské aplikace

Rozšíření na straně aplikace odpovídá fungováním rozhraní návrhu ze sekce 4.11. Chtěl jsem uživateli umožnit výběr úloh k exportování z libovolného výpisu (například z určité skupiny) a vyhnout se přidávání dalšího výpisu na stránku přenosu dat. Proto je seznam vybraných úloh opět ukládán globálně do úložiště Vuex.

Pro čtení obsahu balíků vybraných uživatelem ještě před nahráním na server slouží knihovna *msgpack-lite*. Díky tomu lze uživateli ihned ukázat úlohy v něm obsažené a zaslat seznam závislostí na kontrolu. Po tuto dobu nebyl soubor zálohy ještě posílán po síti. Teprve pokud server oznámí, že všechny závislosti byly nalezeny, povolí se na stránce tlačítko pro nahrání souboru.

Kapitola 6

Testování a experimenty

V této kapitole se budu věnovat testování mnou implementovaných rozšíření a úprav. Testování bylo zaměřené primárně na zkoušení funkčnosti částí rozšíření. Zároveň jsem hledal případné příležitosti pro úpravu uživatelského rozhraní nebo samotného způsobu použití těchto funkcí. Popisuji zde své nálezy a jejich řešení, ovšem implementace popsaná v předchozí kapitole již odráží výslednou podobu systému. Také zde popisuji průběh experimentu, kterým jsem ověřoval správné fungování dávek a porovnával je se souběžným spouštěním úloh.

6.1 Testy funkčnosti

Pro každé z rozšíření jsem prováděl postupně různé akce, které se očekávají od uživatele systému v běžném provozu. Sledoval jsem, jak se samotná funkce i celý systém chová a zda nedochází k chybám. Nalezené chyby, pokud se nějaké vyskytly, jsem následně opravil.

U většiny prováděných úkonů nedošlo k žádným problémům. Několik součástí však v určitých situacích vykazovalo chování nevhodné nebo v rozporu s návrhem. Některé z nich vyžadovaly větší zásahy do doposud implementované funkcionality. Právě tyto problémové oblasti zde chci zdůraznit.

Dávkové spouštění

U systému dávek se ukázala jako problémová implementace samotného spouštění následujících úloh. V prvotní implementaci bylo spuštění navázáno na úplné dokončení předchozí úlohy a nepočítalo ani s jinými koncovými stavy, jako je nenalezení hesla nebo chyba. Po další diskuzi s ostatními členy týmu se navíc ukázalo jako mnohem výhodnější řešení spouštět další úlohu již ve stavu dokončování, tedy po přidělení posledních pracovních jednotek.

Výsledná změna vyžadovala rozšíření použité databázové procedury a zásah i do implementace generátoru. Ve výsledku procedura úplně nahrazuje veškeré dotazy na změny stavu běžících úloh.

Úpravy šablon úloh

V návrhu pro šablony úloh jsem uvedl, že editace šablony bude probíhat použitím formuláře zadání úlohy a uložením pod stejným názvem. Tato funkcionalita však v systému chyběla, protože jsem původně vytvořil koncový bod pouze pro přidání šablony metodou *POST*.

Aby systém fungoval podle návrhu, změnil jsem logiku koncového bodu a upravil jeho metodu na *PUT*, značící možnost úpravy existující položky. Dále jsem upravil dialog vytvoření šablony v aplikaci tak, aby vždy předvyplnil název šablony názvem té vybrané a jasněji dával najevo, že takto se šablona přepíše.

Přiřazování úloh do skupin

I když se v seznamu úloh nezobrazovaly, vznikaly přiřazením již ve skupině obsahovaných úloh ve spojovací tabulce duplikáty. Vedlejším efektem pak bylo nejen nesprávné číslo počtu úloh v bočním panelu, ale i chyba na serveru při pokusu tyto úlohy ze skupiny odebrat.

Jednoduchým zabezpečením před tímto problémem bylo úlohy přidávané do skupiny z této nejprve odebrat. Jelikož se jedná o jednoduché přiřazení bez dalších dat vázaných na relaci, nehrozí ztráta informací.

Úlohy v rozšířeném systému oprávnění

Po rozšíření systému oprávnění uživatelů o pojem vlastníků úloh se v krajních případech nesprávně vypisovaly úlohy. Pokud byl uživatel zadán jako vlastník, ale neměl přiřazená ostatní oprávnění, neměl k těmto úlohám přístup. Podle návrhu však vlastnictví předchází jednotlivá oprávnění.

Ačkoliv se jedná o případ, který by při běžném používání neměl nastat, protože se při vytvoření úlohy přidělí tvůrci všechna oprávnění, chtěl jsem přednost vlastnictví zachovat jako pojistku. K modelu tabulky s oprávněním jsem tedy přidal vlastnosti vracející správnou hodnotu na základě obou dílčích hodnot.

6.2 Experiment – dávkové spouštění

Cílem experimentu je ověřit funkčnost dávkového spouštění úloh a srovnat jeho průběh se spuštěním všech úloh souběžně, viz část 4.7. Experiment proběhne nad sadou připravených úloh uvedených v tabulce 6.1. Úlohy jsou vytvořeny tak, aby některé mohl za krátkou dobu vyřídit i jeden samotný uzel, zatímco jiné vyžadují několik minut práce a ideálně budou systémem rozděleny mezi uzly.

Název	Typ útoku	Typ heše	Možný počet hesel	Prolomitelná hesla
eggplant	kombinační	SHA-384	12 278 061 256	unextendedeggplant
dead	slovník	SHA1	8 171 893 548	zenith
nuke	maska	SHA1	9 273 368 000 000	w@wZ3X209
futile	kombinační	SHA1	266 224 008 961	–

Tabulka 6.1: Základní údaje úloh použitých v experimentu

Serverovou část systému obsluhuje virtuální stroj (Google Compute Engine e2-medium) se 2 vCPU a 4 GB operační paměti pod operačním systémem Ubuntu Linux. Tento server nemá grafický adaptér a neúčastní se lámání jako výpočetní uzel. Připojené uzly jsou umístěny v různých sítích. Jejich specifikace je uvedena v tabulce 6.2. Operační systém všech uzlů je Windows 10 verze 1909.

Jméno stanice	CPU	GPU	RAM
Dia	AMD Ryzen 5 3600	GeForce RTX 2070 SUPER	16 GB
DESKTOP-QS8AR39	AMD Ryzen 5 1600	GeForce GTX 1070	16 GB
utkn	Intel Core i5-6400	Radeon RX580	16 GB

Tabulka 6.2: Specifikace připojených výpočetních uzlů

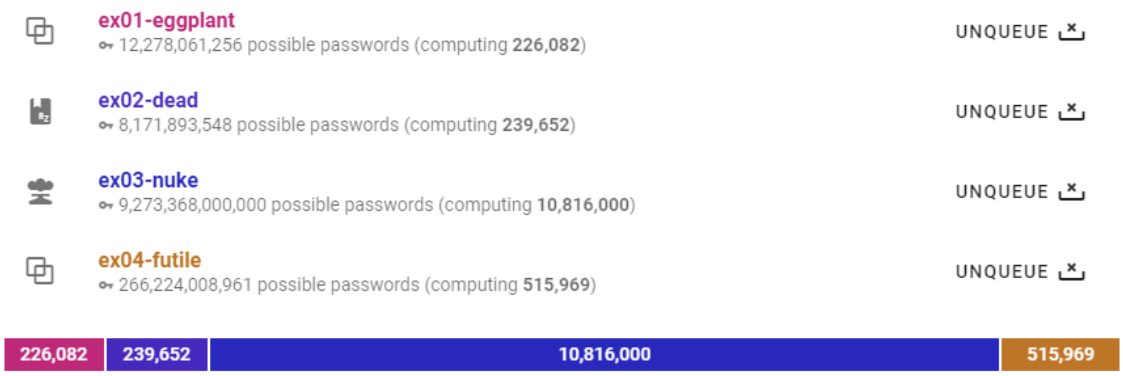
Očekávané chování

V rámci samotné dávky očekávám, že po spuštění proběhnou všechny úlohy v pořadí, v jakém jsou uvedeny ve frontě. Navazující úlohy by se měly spouštět, jakmile předchozí úloha dojde do stavu dokončovaní, případně rovnou skončí. Dávka by se neměla v průběhu vykonávání sama zastavit. Nikdy by v jednu chvíli neměly běžet dvě nebo více úloh, pro které se ještě vytváří pracovní jednotky (stav *running*). Protože se návaznost dávek řeší až na úrovni databáze, neočekávám jakýkoliv znatelný dopad na výkon způsobený režií spouštění následníků.

Ve srovnání se souběžným spuštěním všech úloh, kde toto nelze ovlivnit, předpokládám, že se spolehlivě dříve dokončí úlohy, které umístím na začátek fronty. Z hlediska výkonu by dávka měla odpovídat souběžnému lámání. Může ovšem docházet ke zdržení, obzvláště v případě krátkých úloh, kdy měření výkonu a další režie spojená s přenosem jednotek na uzly pozdrží spuštění další úlohy.

Průběh experimentu

Na začátku jsem připravené úlohy exportoval do souboru, abych je mohl vždy snadno přidat do systému pro další běh a mohl si ponechat předchozí výsledky. Lámání jsem několikrát opakoval jak v dávce tak souběžným spuštěním. Vždy jsem sledoval stav a také měřil čas nezávisle na ukazatelích systému. Dávky byly z importovaných úloh vytvořeny vždy s frontou, kterou zachycuje obrázek 6.1.



Obrázek 6.1: Fronta úloh pro experimentální dávky

Samotné lámání probíhalo dle očekávání. Úlohy v dávce se spouštěly postupně, vždy při přechodu do stavu dokončování, jak je vidět na obrázku 6.2. Rovněž ukazatele průběhu v detailu dávky a souhrnné grafy fungovaly správně. Ukazatele průběhu během zpracovávání zachycuje obrázek 6.3. Graf zobrazující rozdělení hešů do pracovních jednotek všech úloh však při výskytu značně rozdílných hodnot ztrácel na čitelnosti. Pracovní jednotky úloh

s velkým počtem hesel způsobily, že menší úlohy na grafu již nebyly dobře rozlišitelné. Toto je možné pozorovat na obrázku 6.4, který zachycuje souhrnné grafy po skončení dávky.

Při zpracování úloh souběžným spuštěním samozřejmě nebyly souhrnné grafy a celkový průběh k dispozici. Proto jsem úlohy umístil do oddělené skupiny a zde sledoval průběh. Brzy po začátku bylo zřejmé, že se tímto způsobem výrazně rychleji dokončí zpracování krátkých úloh s malým počtem hesel. Generátor totiž může rovnou přidělit pracovní jednotky z obou úloh, namísto aby vždy čekal na jejich dokončení. Zbytek lámání pak probíhal normálně.

Způsob zpracování	**dávkou**	**souběžně**
eggplant	1:29	2:14
dead	1:32	2:14
nuke	3:59	5:14
futile	2:42	6:08
Reálný čas	**8:30**	**6:08**

Tabulka 6.3: Srovnání času lámání mezi dávkovým a souběžným zpracováním

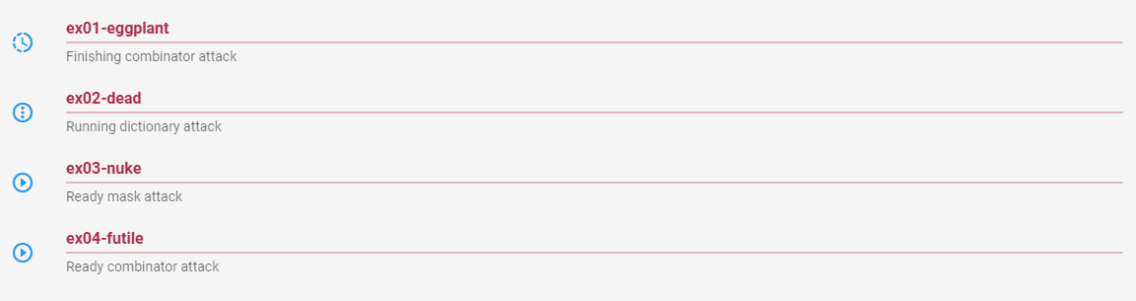
V tabulce 6.3 lze vidět porovnání naměřeného času potřebného k lámání při každém způsobu zpracování. Časy uvedené pro jednotlivé úlohy udávají dobu, po kterou každá úloha běžela. *Reálný čas* pak udává skutečnou dobu od spuštění dávky či úloh až po dokončení poslední úlohy. V případě souběžného zpracování odpovídá nejdéle běžící úloze. Doba zpracování dávky je pak o něco kratší, než součet dílčích časů, protože se následné úlohy spouští již při dokončování té předchozí.

Zhodnocení

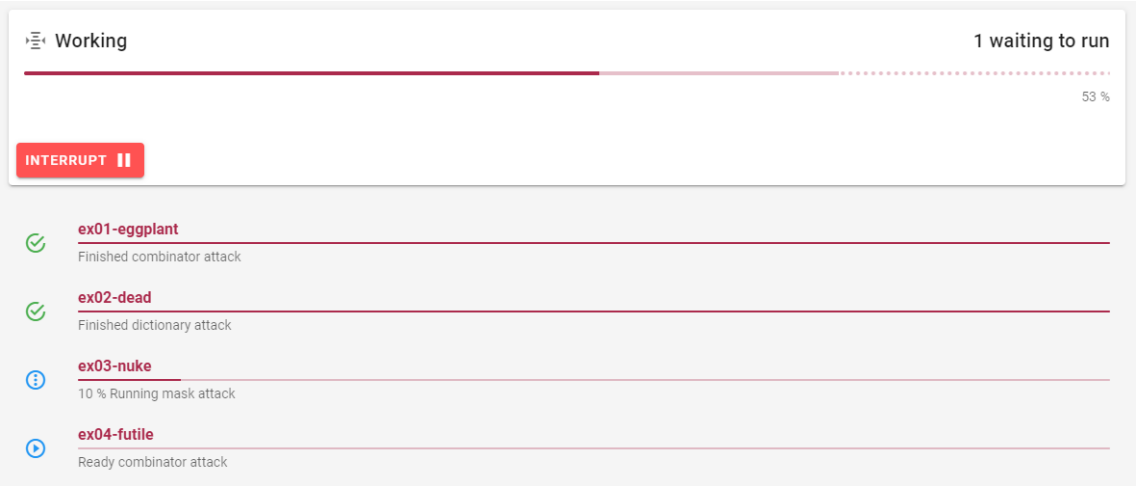
Z pozorovaného chování a naměřených časů lze vyvodit několik zjištění. V první řadě lze konstatovat, že systém dávek funguje podle očekávání. Jak se ale ukázalo, není příliš vhodný pro zpracovávání úloh s malým počtem kandidátních hesel. Jeho primární výhody ovšem platí pro všechny potenciální konfigurace: Uživatel má v dávkách možnost nastavit pořadí úloh podle priority a sledovat průběh a statistiky celé fronty na jednom místě.

Z naměřených časů v tabulce 6.3 pak lze pozorovat, že při souběžném zpracování trval běh všech dílčích úloh déle, než v případě dávky. Jelikož se v tomto případě dělí pracovní jednotky ze všech úloh, je toto očekávaný důsledek. I když celkový čas strávený lámáním je kratší, nelze ovlivnit pořadí zpracování. Například úloha *futile* tak byla pozdržena ostatními a dokončila se až jako poslední. Při dávkovém zpracování lze dílčí časy uvažovat i v jiném uspořádání fronty. V případě, že by tato úloha měla pro uživatele větší význam, mohl by jí zařadit na začátek fronty a očekávat podobný čas necelých tří minut.

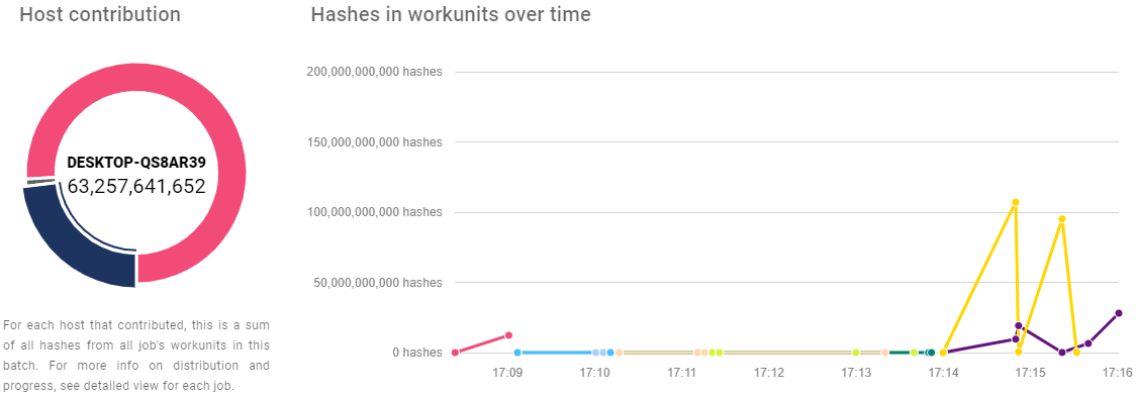
Graf rozdělení hešů, jak je popsáno výše, nedokáže znázornit rozdíly u malých jednotek, pokud se v něm vykytují výrazně větší hodnoty. Tento problém by mohlo vyřešit použití logaritmické stupnice na dané ose. Aktuálně používaná knihovna na vykreslování grafů ovšem toto nepodporuje.



Obrázek 6.2: Spuštěná následující úloha s předchozí ve stavu dokončování



Obrázek 6.3: Ukazatele průběhu celé dávky a dílčích úloh během lámání



Obrázek 6.4: Grafy experimentální dávky po dokončení lámání

6.3 Testy s uživateli

I když uživatelské testy jako takové jsem neprováděl, považuji zde za jejich náhradu, neli lepší alternativu, neustálý náhled uživatelů pracujících se systémem. Toho se mi dostalo díky práci s ostatními členy vývojářského týmu systému Fitcrack. Protože se jedná o úpravy a rozšíření již existující, testované aplikace, nejvíce jsem ocenil právě vstupy a možnost diskuze s lidmi, kteří dokážou nejlépe posoudit, jak by se měly nové součásti chovat.

V závěrečné fázi vývoje pak mělo několik zájemců, používajících jiná řešení pro lámání hesel, možnost vyzkoušet si předběžné vydání systému. Uživatelské rozhraní administrační aplikace hodnotili velmi pozitivně. Webovou aplikaci označili za intuitivnější a vizuálně přívětivější, než podobná, komerčně dostupná řešení. Po testování produktu kriminální policií, vrchní komisař kpt. Mgr. Lukáš Lazecký (Služba kriminální policie a vyšetřování Krajského ředitelství policie Jihomoravského kraje v Brně) řekl:

> *„Děkujeme za možnost vyzkoušet.*
>
> *Nástroj působí velmi přehledně a prakticky. Pro svůj účel nabízí dostatečné a přehledné volby. V minulosti jsme při použití komerčních produktů narazili na komplikace při inicializaci úkolu, přičemž ne všechny nabízené možnosti a funkce byly srozumitelně popsané.*
>
> *Pro praktické použití bychom pro případ scénáře, kdy je známá představa o hledaném hesle nebo jeho části uvítali podrobnějšího průvodce při tvorbě masky hledaného řetězce obsahující např. příklady a souběžně orientační informaci o přibližné době pro ukončení úlohy, tak aby bylo možné parametry hledané masky ještě před spuštěním úlohy modifikovat za účelem dosažení reálné doby pro hledání výsledku.“*

Stojí za zmínku, že některé zmíněné náměty byly již do systému zapracovány. Například odhad času potřebného k lámání byl v rámci vylepšování systému přepracován a zpřesněn. Další návrhy, jako průvodce tvorbou masky, který by zobrazoval příklady hesel, by bylo vhodné implementovat v rámci budoucího zlepšování produktu.

Kapitola 7

Závěr

Cílem práce a předmětem této zprávy bylo rozšíření administrační aplikace systému Fitcrack o nové způsoby práce s lámacími úlohami. Mimo to jsem ale v rámci práce i modernizoval celé rozhraní administrační aplikace. Navrhl a zapracoval jsem systém pro tvorbu a aplikaci šablon úloh, které umožňují při jejich zadávání předvyplnit formulář. Rozšířil jsem možnosti organizace úloh o možnost třídění do skupin a přepracoval jsem zobrazení výpisu a detailního zobrazení úloh. Přidal jsem možnosti, jak pracovat s více úlohami současně a také vytvářet fronty úloh pro spouštění v dávkách. Navíc jsem navrhl a implementoval funkcionalitu exportování úloh do souboru a jejich opětovné nahrání do systému. Celý systém administrace jsem také více zabezpečil implementací pokročilých možností přidělování oprávnění jednotlivým uživatelům.

Brzy po začátku práce na tomto projektu jsem se stal členem týmu vyvíjejícího Fitcrack. Výsledný podsystém administrace pro Fitcrack významně ovlivnil vstup od zbytku týmu. Návrhy, jak na samotná rozšíření, tak i jejich vizuální podobu a fungování, se v průběhu času stále vyvíjely. Díky tomu, že se v týmu používala právě vyvíjená verze systému, jsem měl stále k dispozici zpětnou vazbu a návrhy od ostatních členů.

V rámci testování svých úprav jsem pak odhalil další nedostatky a chyby, na které jsem tak měl šanci reagovat a systém doladit. Opravil jsem po testování několik neduhů v rámci mých rozšíření a také kompletně přepracoval mou původní implementaci zpracovávání dávek. Experiment s dávkovým spouštěním následně ukázal, že systém dávek již funguje správně, i když nemusí být z časového hlediska pro všechna zadání výhodný, zejména pokud jde o časově nenáročné úlohy.

Přesto má systém Fitcrack, jak v rámci administrace, tak jako celek, stále velký potenciál na rozšiřování a úpravy. Jednak v rámci doplňkové funkcionality, jako jsou uživateli navržené příklady hesel u masek nebo kontextová nápověda, průvodce aplikací a lepší provázání s manuálem. Další potenciál na zlepšení pak vidím stran výkonu administračního systému. Některé datové struktury vracené koncovými body obsahují spoustu zanořených informací, které aplikace ani nevyužije. Zkrátit čas k prvnímu zobrazení některých sekcí v aplikaci by mohlo pomoci oddělené asynchronní načítání informací pro komponenty. Grafy v aplikaci při vykreslování většího počtu dat také značně ztrácí výkon. Bylo by vhodné přílišné množství dat například agregovat do bodů s průměrnou hodnotou. Proto bych se systému rád nadále v určitém rozsahu věnoval, alespoň jako dobrovolník udržující repozitář ve službě Github.

Literatura

[1] Anderson, D. P. BOINC: A system for public-resource computing and storage. In: *Proceedings - IEEE/ACM International Workshop on Grid Computing*. 2004, s. 4–10. DOI: 10.1109/GRID.2004.14. ISBN 0769522564.

[2] Atanassov, R., Atkins, T. Jr., Etemad, E. a Baron, D. *CSS Flexible Box Layout Module Level 1*. Candidate Recommendation. W3C, listopad 2018. Dostupné z: https://www.w3.org/TR/2018/CR-css-flexbox-1-20181119/.

[3] Bray, T. *The JavaScript Object Notation (JSON) Data Interchange Format* [RFC 7159 (Proposed Standard)]. 7159. Internet Engineering Task Force, březen 2014. Dostupné z: http://www.ietf.org/rfc/rfc7159.txt.

[4] Fielding, R. T. a Taylor, R. N. *Architectural styles and the design of network-based software architectures*. Irvine, CA, 2000. Disertační práce. University of California.

[5] Geissler, G. L., Zinkhan, G. M. a Watson, R. T. The influence of home page complexity on consumer attention, attitudes, and purchase intent. *Journal of Advertising*. 2006, sv. 35, č. 2, s. 69–80. DOI: 10.1080/00913367.2006.10639232. ISSN 00913367.

[6] Hranický, R., Lištiak, F., Mikuš, D. a Ryšavý, O. On Practical Aspects of PCFG Password Cracking. In: *Data and Applications Security and Privacy*. Springer Verlag, 2019, sv. 33, č. 11559, s. 43–60. DOI: 10.1007/978-3-030-22479-0_3. ISBN 978-3-030-22478-3. Dostupné z: https://www.fit.vut.cz/research/publication/11955.

[7] Hranický, R., Zobal, L., Ryšavý, O. a Kolář, D. Distributed password cracking with BOINC and hashcat. *Digital Investigation*. 2019, sv. 30, č. 1, s. 161–172. DOI: 10.1016/j.diin.2019.08.001. ISSN 1742-2876. Dostupné z: https://www.fit.vut.cz/research/publication/11961.

[8] Hranický, R., Zobal, L., Večeřa, V. a Múčka, M. *The architecture of Fitcrack distributed password cracking system*. FIT-TR-2018-03, Brno, CZ. Vysoké učení technické v Brně, Fakulta informačních technologií, 2018. 61 s. Dostupné z: https://www.fit.vut.cz/research/publication/11887.

[9] Múčka, M. *Webová aplikace pro vzdálenou správu systému Fitcrack*. Brno, CZ, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21121/.

[10] Oracle. *MySQL 5.0 Reference Manual*. 2016. [Online; navštíveno 9.5.2020]. Dostupné z: https://downloads.mysql.com/docs/refman-5.0-en.pdf.

[11] Pokorný, Š. *Analytické zpracování metadat k lámání hesel*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21745/.

[12] Van Rossum, G. *Python tutorial*. CS-R9526, Amsterdam, NL. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[13] You, E. *Vue.js: a (re)introduction*. Listopad 2015. [Online; navštíveno 27.12.2019]. Dostupné z: https://medium.com/@youyuxi/vue-js-a-re-introduction-ed159414a1be.

Příloha A

Obsah přiloženého paměťového média

Přiložený disk obsahuje:

- tento dokument ve formátu PDF,
- zdrojové soubory této zprávy (LaTeX),
- kompletní zdrojové kódy systému Fitcrack,
- uživatelský manuál ve formátu PDF.

Zdrojové kódy jsou také v repozitáři na adrese https://github.com/nesfit/fitcrack. **Součástí zdrojových souborů je i návod na instalaci produktu.** Mé úpravy a rozšíření se týkají konkrétně následujících adresářů:

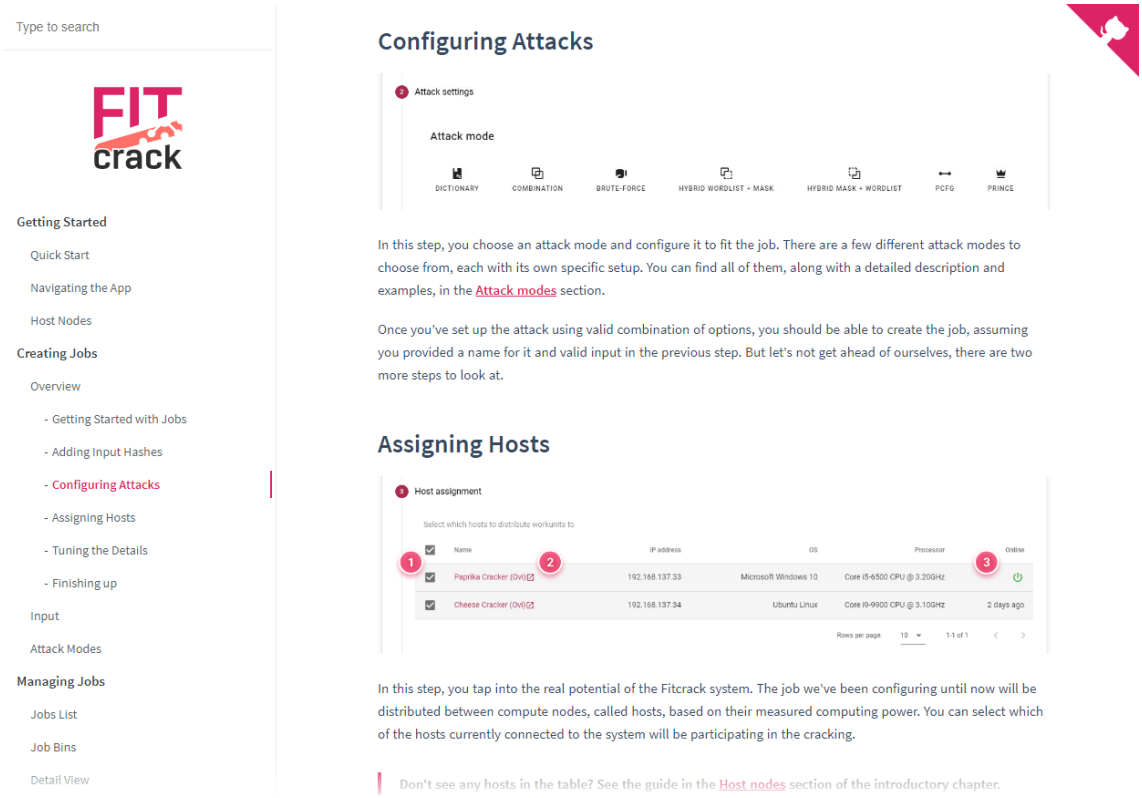
- `/webadmin/fitcrackFE` – Webová aplikace administrace
- `/webadmin/fitcrackAPI` – Serverová část administrace
- `/server/sql` – Skripty SQL pro založení databáze
- `/server/src` – Zdrojové kódy generátoru a asimilátoru
- `/docs` – Uživatelský manuál pro web

Podrobný výpis souborů, které jsem vytvořil nebo upravoval, se nachází na přiloženém disku v souboru `fitcrack/upravene_soubory.txt`. Historie mých změn je pak dohledatelná v repozitáři ve službě Github, například v seznamu na adrese https://github.com/nesfit/fitcrack/commits?author=OviOvocny.

Příloha B

Manuál

Uživatelský manuál jsem psal jako samostatný dokument primárně pro prezentaci na webu. Manuál popisuje používání systému skrze administrační aplikaci. Jedná se o text mířený na nové uživatele. Seznamuje postupně se základy práce se systémem a představuje jednotlivé funkce aplikace. Webová prezentace je dostupná na adrese `https://nesfit.github.io/fitcrack` nebo lze spustit lokálně z adresáře `/docs` v repozitáři pomocí `npx docsify serve` (vyžaduje Node.js). Podoba webové verze je na obrázku B.1. Manuál lze pro potřeby publikace převést také do dokumentu PDF. V této podobě je také dostupný na přiloženém paměťovém médiu, ovšem webová verze je preferovaná.



Obrázek B.1: Úryvek manuálu na webu