Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Interaktívny pracovný hárok pre výučbu logiky pre informatikov

Bakalárska práca

2020
Nikolaj Kniha

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Interaktívny pracovný hárok pre výučbu logiky pre informatikov

Bakalárska práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	Aplikovaná informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	Mgr. Ján Kľuka, PhD.

Bratislava, 2020
Nikolaj Kniha

Poďakovanie: Tu môžete poďakovať školiteľovi, prípadne ďalším osobám, ktoré vám s prácou nejako pomohli, poradili, poskytli dáta a podobne.

Abstrakt

Bakalárska práca sa zaoberá vývojom webovej aplikácie, ktorá umožní študentom riešiť úlohy zadávané na predmete Matematika 4. Aplikácia umožňuje používateľovi prechádzať vlastné repozitáre a súbory v nich uložené na stránke GitHub, vytvárať nové zadania a priestor na riešenie vo forme univerzálnej karty. Každá z kariet je jedného z dvoch typov: textová, ktorá slúži iba na uloženie textu zadania alebo odpovede a riešičová, ktorá umožňuje využitie jednej z implementovaných aplikácií, zameraných na výučbu matematickej logiky. Dáta z kariet sú následne ukladané na GitHub a pri ďalšom otvorení rovnakého súboru so zadaniami opäť načítané. Pre naprogramovanie aplikácie sú primárne využité knižnice React a Redux.

Kľúčové slová: webová aplikácia na strane klienta, matematická logika

Abstract

Abstract in the English language (translation of the abstract in the Slovak language).

Keywords:

Obsah

Úvod **1**

1 Webové aplikácie a technológie **2**
1.1 Webová aplikácia 2
1.2 HTML5, CSS3, JavaScript 2
1.3 React 3
1.3.1 Stavový vs funkcionálny komponent 3
1.3.2 Využitie komponentov 4
1.3.3 Výhody využívania Reactu 5
1.4 Redux 6
1.5 Bootstrap 4 7
1.6 Markdown 8
1.7 Node.js 8
1.8 npm 9
1.9 Firebase 9
1.10 REST API 9
1.11 Github 10

2 Použité bakalárske práce **11**
2.1 Prieskumník sémantiky logiky prvého rádu 11
2.2 Educational tools for first order logic 11
2.3 A proof assistant for first-order logic 12

3 Existujúce riešenia **14**
3.1 Jupyter Notebook 14
3.2 Wolfram Mathematica 15
3.3 MATLAB 16

4 Ciele práce **18**
4.1 Účel aplikácie 18
4.2 Požiadavky na funkčnosť aplikácie 18

4.2.1 Používateľ 18
4.2.2 Prehliadač repozitárov 19
4.2.3 Prehliadač zadaní 19

5 Návrh **21**
5.1 Používateľské prostredie 21
5.1.1 Neprihlásený používateľ 21
5.1.2 Prihlásený používateľ 21
5.1.3 Prehliadanie zadaní 22
5.1.4 Práca na zadaní 23
5.2 Ukladanie dát 23
5.2.1 Krátka životnosť 23
5.2.2 Stredná životnosť 25
5.2.3 Dlhá životnosť 25
5.3 Štruktúra aplikácie 27
5.4 Riešiče 28

6 Implementácia **29**
6.1 Štruktúra aplikácie 29
6.2 Práca s dátami 30
6.3 Tok dát 30
6.3.1 Redux 30
6.3.2 LocalStorage 31
6.3.3 GitHub 31
6.4 Prihlásenie a odhlásenie používateľa 31
6.5 Prehľadávanie repozitárov 32
6.5.1 Získavanie dát 33
6.5.2 URL pre pohyb v repozitároch 33
6.6 Úlohy 35
6.6.1 Získavanie a odosielanie dát 35
6.7 Integrácia riešičov 36
6.7.1 Tableau Editor 36
6.7.2 Resolution Editor 38

7 Testovanie **40**
7.1 Cieľ testovania 40
7.2 Priebeh testovania 40
7.3 Výsledky testovania 40

Záver **41**

Zoznam obrázkov

1.1 Dátový tok v Reduxe. 6
1.2 Príklad komponentu Card v Reacte podľa kódu 11 7
1.3 Príklad textu napísaného v Markdown. 8
1.4 Webové užívateľské prostredie Firebase 10

2.1 Prieskumník sémantiky logiky prvého rádu 12
2.2 Educational tools for first order logic 13
2.3 A proof assistant for first-order logic 13

3.1 Jupyter Notebook interface 15
3.2 Wolfram Mathematica Interface 16
3.3 Matlab interface 17

5.1 Stránka neprihláseného používateľa 22
5.2 Stránka prihláseného používateľa 23
5.3 Zoznam zadaní 24
5.4 Práca so zadaním 24
5.5 Architektúra aplikácie pre prácu s dátami 26
5.6 Diagram uloženia dát v stave Redux Store 27

Zoznam tabuliek

Úvod

V súčasnosti sa viac a viac ľudí zameriava na digitalizáciu a zlepšenie pracovného prostredia pre študentov aj učiteľov vytváraním nástrojov pre podporu vyučovania. Študenti matematickej logiky v súčasnosti riešia praktické aj teoretické úlohy, tie praktické sú riešené a odovzdávané v digitálnej podobe, no teoretické stále v papierovej, s možnosťou využitia samostatných aplikácií zameraných na zlepšenie chápania a získanie skúseností pri riešení úloh na tomto predmete. Všetky tieto aplikácie fungujú v súčasnosti samostatne a slúžia iba na podporu vzdelávania.

Cieľom tejto bakalárskej práce je vytvoriť webovú aplikáciu zameranú na zlepšenie a digitalizáciu výučbového procesu pre teoretické úlohy, ktorá bude mať formu interaktívneho pracovného hárku. Táto aplikácia bude slúžiť ako študentom aj vyučujúcim pri zadavaní, respektíve riešení domácich úloh.

Pre zlepšenie kvality výučby je cieľom práce implementovať predchádzajúce bakalárske práce, čiže aplikácie na riešenie rôznych matematických úloh. Tieto riešiče budú zakomponované v našej aplikácii tak, aby bolo možné ich výsledky priamo uchovávať a znova načítať do aplikácií pre účely hodnotenia.

Motiváciou pre tvorbu tejto aplikácie je uľahčenie práce vyučujúcim ako aj zvýšenie kvality výučby pre študentov. Doteraz boli riešiče málo využívané a žiadna aplikácia neponúkala riešenie teoretických úloh na jednom mieste, preto táto práca spája funkčnosť a jednoduchosť pracovného hárku obohatenú o možnosť využívať rôzne výpočtové nástroje.

V prvej kapitole sa zaoberáme využitými technológiami, systémami s podobnou funkčnosťou, ktoré nám slúžili aj ako predloha a predchádzajúcimi bakalárskymi prácami, ktoré sa zaoberali implementovanými riešičmi. Druhá kapitola obsahuje stručné a prehľadné zhrnutie cieľov práce a požiadaviek na jej funkcionalitu. V tretej kapitole sa venujeme podrobnému návrhu používateľského prostredia, spôsobu ukladaniu dát a práci s nimi, štruktúre aplikácie a možnostiam na zapojenie riešičov. Štvrtá kapitola popisuje detaily ohľadom implementácie aplikácie a jej komponentov. Posledná kapitola ponúka výsledky testovania, ktoré prebehlo priamo na študentoch predmetu Matematika 4.

Kapitola 1

Webové aplikácie a technológie

1.1 Webová aplikácia

Webová aplikácia je aplikácia, ktorá je uložená na vzdialenom serveri, pomocou internetu poskytnutá klientovi a pomocou webového prehliadača zobrazená užívateľovi, čiže ide o aplikáciu typu klient-server.

Webové aplikácie môžu byť navrhnuté pre rôzne využitia a prístupné komukoľvek s pripojením na internet, v prípade firemných aplikácii s pripojením na intranet. Existuje niekoľko druhov webových aplikácií, tá naša je typu rich client application (RIA). Aplikácie typu RIA ponúkajú vlastnosti a funkcionalitu tradičných aplikácií, ktoré vyžadujú inštaláciu na zariadenie klienta. RIA aplikácie rozdeľujú prácu medzi klienta a aplikačný server. Tento prístup umožňuje vykonávať úpravy, kalkulácie a rôzne akcie na strane klienta a tým znížiť komunikáciu medzi klientom a serverom. [1]

Rozdiel oproti iným webovým aplikáciam je v mieste, kde sa vykonáva logika aplikácie. Pri jednoduchých webových aplikáciach sa aplikačná logika vykonáva na strane servera, pri RIA aplikácii je to však na strane klienta.

1.2 HTML5, CSS3, JavaScript

HTML5 (HyperText Markup Language) je značkovací jazyk využívaný na tvorbu webových stránok. Umožňuje zapisovať dokumenty obsahujúce text, hypertextové odkazy, skripty a ďalšie.

CSS (Cascading Style) je jazyk slúžiaci na popísanie zobrazenia elementov v dokumentoch typu HTML alebo XML. Umožňuje teda meniť vzhľad a rozmiestnenie elementov v dokumente.

JavaScript je multiplatformový a objektový skriptovací jazyk, ktorý umožňuje implementáciu komplexných funkcií prevažne na internetových stránkach. Väčšinou sa vkladá priamo do HTML kódu stránku a o interpretáciu tohto kódu sa zvyčajne stará

internetový prehliadač klienta. Slúži hlavne na ovládanie dynamických prvkov stránky (tlačidlá, textové polia, atď.). JavaScript slúžil výlučne pre programovanie na strane klienta, to sa však postupom času zmenilo a stal sa používaným programovacím jazykom aj na strane servera, kde sa využíva prevažne Node.js.

1.3 React

React je JavaScriptová knižnica, ktorá slúži na efektívne vytváranie používateľských rozhraní. Využíva deklaratívny prístup, vďaka čomu sú v nej vytvorené aplikácie efektívne, flexibilné a umožňuje aktualizovať iba tie komponenty, ktorých dáta boli zmenené. React sa dá využiť ako základ pre vývoj webovej alebo mobilnej „single-page“ aplikácie. Pretože React sa stará iba o renderovanie dát v DOM, na správne fungovanie celej aplikácie sú potrebné ďalšie knižnice na správu jeho stavu, routing a interakciu s API. Výhodou Reactu je spolupráca s už existujúcim kódom a je jednoduché používať ho s knižnicou jQuery, frameworkom Angular atď.[2]

V Reacte máme možnosť vytvárať komponenty, ktoré sú zodpovedné za vykresľovanie elementov v jazyku HTML. Výhodou je však znovupoužiteľnosť týchto komponentov a možnosť vnárať ich do seba. Pri programovaní v Reacte sa snažíme o vytváranie menších samostatných komponentov, ktoré budú použité v dynamickej webovej aplikácii. Vytvorila a spravuje ho spoločnosť Facebook spolu s komunitou samostatných vývojárov.

1.3.1 Stavový vs funkcionálny komponent

React nám umožňuje vytvárať dva druhy komponentov, funkcionálny a triedny komponent. Oba typy môžu prijať informácie, ktoré im boli poslané a pristupujú k nim pomocou premennej Props, v ktorej sa môžu nachádzať dáta aj funkcie, ktoré potom komponent využíva.

Triedny komponent má zložitejšiu syntax a vytvára funkciu render(), ktorá slúži na vykreslenie elementov.

```
1  class Osoba extends React.Component {
2      render(){
3          return (
4              <div>
5                  <h1>Ahoj {this.props.meno}</h1>
6              </div>
7          )
8      }
9  }
```

Listing 1.1: Syntax triedneho komponentu

Funkcionálny komponent má jednoduchšiu syntax, pretože je to čisto JavaScriptová funkcia, ktorá vracia Reactový element.

```
1     const Osoba = (props) => {
2         return (
3             <div>
4                 <h1>Ahoj {props.meno}</h1>
5             </div>
6         )
7     }
```

Listing 1.2: Syntax funkcionálneho komponentu

Výhodou triedneho komponentu je možnosť využívať stav a Lifecycle Hooks. Stav komponentu je jednoduchý objekt slúžiaci na ukladanie dát, zvyčajne sa označuje ako state, môže však mať aj iný názov. V prípade, že sa dáta v stave zmenia, funkcia render() znova prekreslí komponent, preto by sa v stave mali nachádzať iba dáta, ktoré priamo ovplyvňujú vykresľovanie komponentu. Lifecycle Hooks sú funkcie, ktoré sa spustia iba ak nastane jedna z určených situácií v komponente, napríklad načítanie komponentu, zmena stavu, nové Props a podobne. Vtedy sa určená funkcia spustí a vieme spracovať nové dáta alebo vykonať požadovanú akciu.

V minulosti bol stav a lifecycle hooks určené výlučne pre triedny komponent. Po novej aktualizácii na verziu React 16.8 s názvom Hooks však prišla možnosť využívať stav pomocou funkcie useState() a lifecycle hooks pomocou funkcie useEffect() aj vo funkcionálnom komponente.

1.3.2 Využitie komponentov

Majme triedny komponent s názvom App, ktorý má v stave uložené počty kusov domácich zvierat. V tomto komponente budeme mať zadefinovanú funkciu, ktorá bude zvyšovať počet kusov konkrétneho zvieraťa. Stav je immutable, takže ho nesmieme meniť priamo. Triedny komponent nám na zmenu stavu ponúka funkciu setState(), ktorej argument bude nový stav. Vo funkcii render budeme vykresľovať vlastný komponent Pocitadlo, ktorý zobrazí počet zvierat a taktiež tlačidlo na zvýšenie ich počtu. Po stlačení tlačidla sa nám zmení stav, to pustí nové prekreslenie komponentu, ktorý tieto dáta používa.

```
1 class App extends React.Component {
2     constructor(props){
3         this.state = {
4         zvierata:[{nazov: "ovca", pocet: 0},
5                 {nazov: "sliepka", pocet: 0}]
6         }
7     }
8
```

```
9     pridajZviera = (nazov) => {
10        this.setState({
11            zvierata: this.state.zvierata.map(item =>
12            (item.nazov === nazov ? {nazov: nazov, pocet: item.pocet +
                  1}: item))
13        })
14    }
15
16    render(){
17      return (
18      <div>
19          {
20          this.state.zvierata.map(item => {
21              return (
22                  <Pocitadlo zviera = {item} pridajZviera = {this.
                        pridajZviera}/>
23              )}
24          }
25      </div>
26      )
27  }
28 }
```

Listing 1.3: Triedny komponent App

```
1     const Pocitadlo = ({zviera, pridajZviera}) => {
2         return (
3             <div>
4                 <h1>{zviera.nazov} : {zviera.pocet}</h1>
5                 <Button onClick={() => pridajZviera(zviera.nazov)} />
6             </div>
7         )
8     }
```

Listing 1.4: Funkcionálny komponent Pocitadlo

Na tomto príklade vidíme viacero dôležitých vlastnosti Reactu. Prvá je stav, v ktorom si môžeme uložiť akýkoľvek typ premennej alebo objekt. Po zmene tejto premennej sa prekreslí komponent, ktorého sa táto zmena týka. Taktiež vidíme možnosť opakovane použiť rovnaký komponent, ktorému len stále posunieme iné dáta.

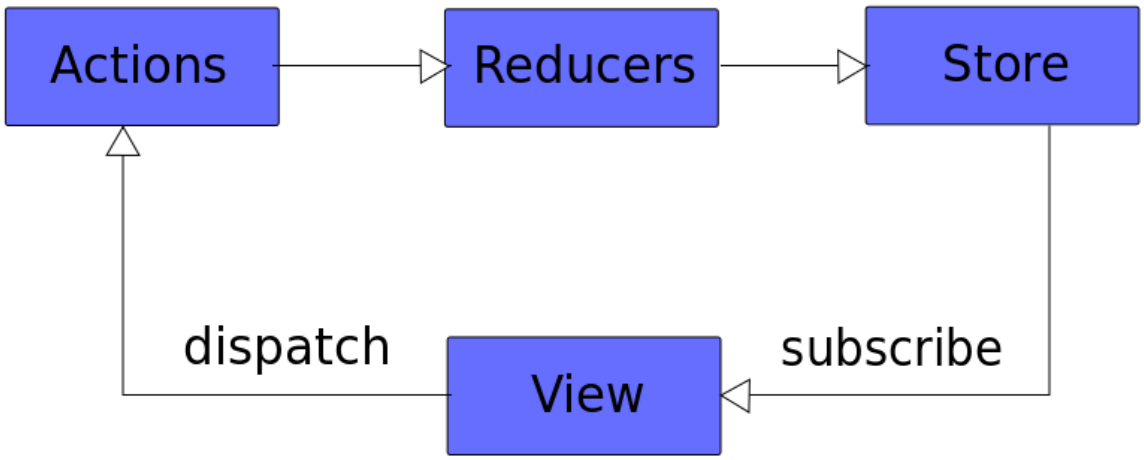
1.3.3 Výhody využívania Reactu

Najväčšou výhodou Reactu je ReactDOM, teda virtuálny DOM vytvorený Reactom. Normálny DOM (Document Object Model) v jednoduchosti reprezentuje UI aplikácie a vždy keď nastane zmena, DOM sa aktualizuje aby zobrazil túto zmenu. DOM má stromovú štruktúru, takže zmena jeho údajov je rýchla, problém je však, že po úprave

sa musí komponent aj so všetkými potomkami znova vyrenderovať a to je pomalé. Tento probl0m rieši Virtuálny DOM, ktorý je virtuálnou reprezentáciu DOM a pri každej zmene sa najskôr aktualizuje virtuálny DOM, ktorý sa porovná so skutočným DOM a vypočíta najlepší spôsob ako vykonať tieto zmeny aby nezabrali zbytočne veľa času a nemuseli sa prerenderovať aj prvky, ktoré sa nezmenili. [5]

1.4 Redux

Redux je open-source JavaScriptová knižnica slúžiaca na manažment stavu aplikácie spustenej vo webovom prehliadači. Stav aplikácie je objekt, v ktorom sú uchovávané dáta, s ktorými aplikácia pracuje a v akom stave sa celá aplikácie v danom momente nachádza. Všetky dáta sú tak uložené na jednom mieste, v stave, kde je k ním jednoduchý prístup.



Obr. 1.1: Dátový tok v Reduxe.

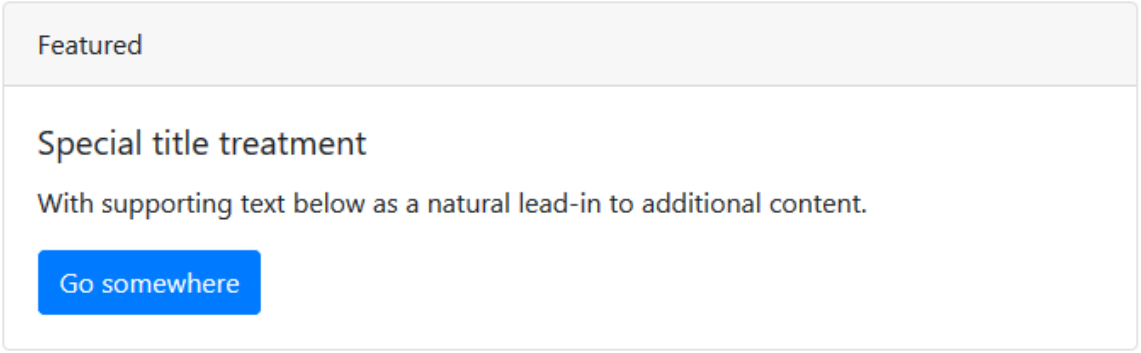
Na obrázku vidíme diagram znázorňujúci tok dát v Reduxe. V Actions sa nachádzajú klasické JavaScriptové objekty, ktoré nesú informácie o tom aká akcia sa vykonala, prípadne ďalšie atribúty a atribút type, ktorý identifikuje vykonanú akciu. Po zavolaní akcie z Reactového komponentu sa zavolá Reducer, čo je funkcia, ktorá prijíma action a state ako argumenty a následne zmení aktuálny stav podľa požadovanej akcie a vráti nový stav, ktorý sa pošle do Store a odtiaľ si ho preberie View čo spôsobí prerenderovanie daného Reactového komponentu podľa požiadaviek. To nám uľahčí spravovanie stavu ako aj oddelenie a sprehľadnenie kódu.

1.5 Bootstrap 4

Bootstrap 4 [10] je open-source CSS framework vytvorený spoločnosťou Twitter, ktorého úlohou je zjednodušiť vytváranie používateľského rozhrania. CSS framework je knižnica, ktorá umožňuje vytvárať jednoduchší a štandardizovaný dizajn webových aplikácií za použitia jazyku CSS. Bootstrap umožňuje urýchliť vytváranie dizajnovej časti aplikácie vďaka komplexnému systému preddefinovaných tried, ktoré sa dajú aplikovať na HTML elementy dokumentu. Na rozloženie elementov využíva systém Grid, čiže rozmiestnenie v mriežke, čo zjednocuje vzhľad rôznych typov prvkov. To umožňuje vytváranie jednotných komponentov používateľského rozhrania, ktoré sa priamo v HTML nenachádzajú - napríklad Karty, Záložky atď. Taktiež sa stará o zobrazovanie na rôznych zariadeniach, pričom zobrazenie elementov môže užívateľ nastaviť ručne alebo nechať Bootstrap aby to urobil automaticky. Pre správne fungovanie Bootstrapu s Reactom bol vytvorený samostatný framework React Bootstrap, vďaka ktorému je možné priamo využívať tieto komponenty.

```
1   <Card>
2     <Card.Header>Featured</Card.Header>
3     <Card.Body>
4       <Card.Title>Special title treatment</Card.Title>
5       <Card.Text>
6         With supporting text below as a natural lead-in to
              additional content.
7       </Card.Text>
8       <Button variant="primary">Go somewhere</Button>
9     </Card.Body>
10  </Card>
```

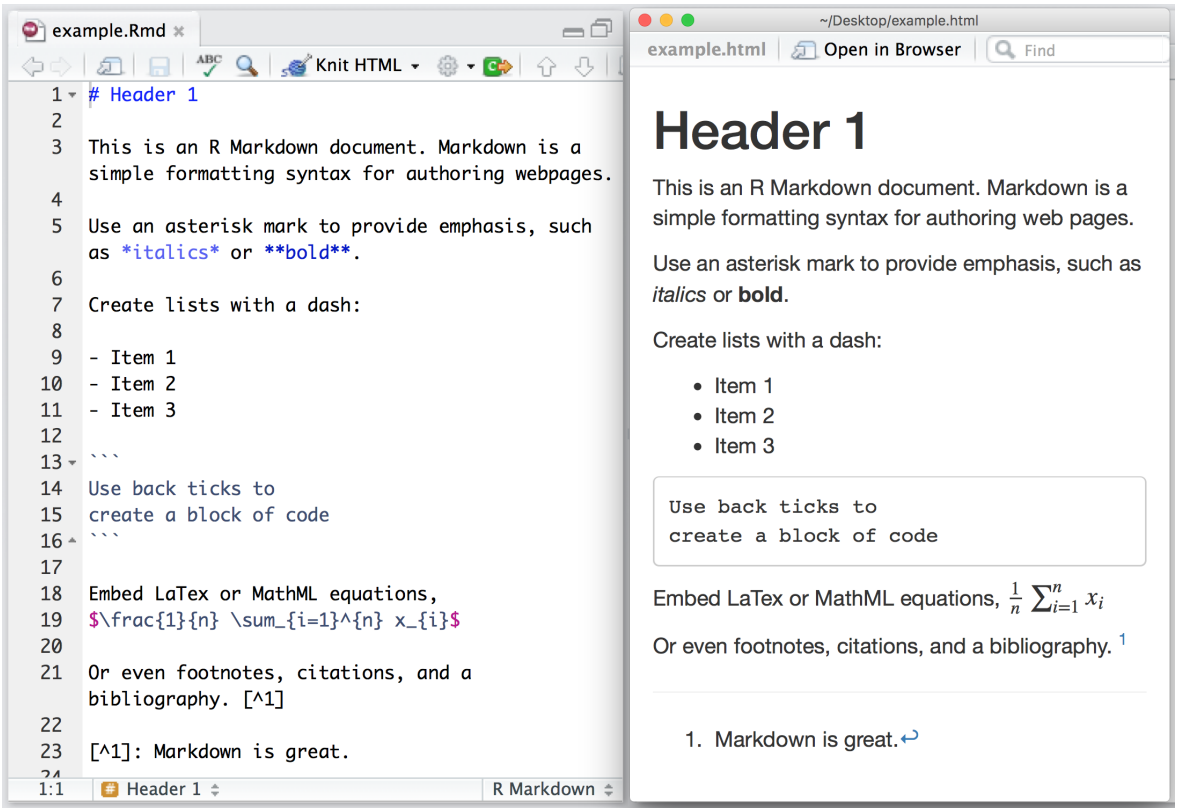
Listing 1.5: Využitie komponentu Card v Reacte



Obr. 1.2: Príklad komponentu Card v Reacte podľa kódu 11

1.6 Markdown

Markdown je odľahčený značkovací jazyk, ktorý slúži na úpravu čistého textu. Jeho dizajn dovoľuje finálny text jednoducho konvertovať do rôznych formátov, v našom prípade do HTML. Často sa používa na formátovanie príspevkov v diskusiách alebo jednoduchých dokumentov. V našom prípade bude použitý v spolupráci s knižnicou React-Markdown [6] na formátovanie zadaní úloh pre študentov.



Obr. 1.3: Príklad textu napísaného v Markdown.

1.7 Node.js

Node.js [2] je open-source multiplatformové prostredie slúžiace na vývoj aplikácií na strane servera v jazyku JavaScript. Slúži na spúšťanie JavaScriptového kódu mimo prehliadača, teda napríklad na strane servera. Vďaka tomu vytvára obsah dynamickej webovej stránky ešte pred tým, než sa pošle klientovi. Taktiež sa používa aj ako interpreter pre programy napísané v JavaScripte, ktoré sú spúšťané z príkazového riadka, napríklad manažment balíkov npm, testovací framerwork Jest alebo prekladač Babel, ktorý sa stará o preloženie novších verzií JavaScriptu do starších, aby správne pracovali aj v starších prehliadačoch.

1.8 npm

npm (Node Package Manager) slúži na sťahovanie a manažment softwérových balíkov potrebných pre náš projekt. Vďaka veľkej databáze dostupných softwérových balíkov je možné rýchlo a efektívne vytvárať a spravovať webové aplikácie.

1.9 Firebase

Firebase [4] je platforma, ktorá slúži na vývoj webových a mobilných aplikácií a je typu typu BaaS. Baas, teda Backend as a Service, znamená, že vývojari sa nemusia zaťažovať s vývojom a spravovaním backendu, čiže ako bude fungovať autentifikácia, databáza atď. Webová či mobilná aplikácia využíva na komunikáciu s Firebase WebSockety, čiže komunikačný protokol, ktorý vytvorí obojsmerný komunikačný kanál medzi aplikáciou a serverom, vďaka čomu je táto komunikácia veľmi rýchla.

Firebase ponúka dve možnosti na výber databázy, staršiu Realtime Database a novú Cloud Firestore. Obe z týchto databáz využívajú NoSQL databázu, no Cloud Firestore je v súčastnosti preferovaná kvôli lepšej bezpečnosti, škálovateľnosti, atď. Súbory v databázach sú teda ukladané ako JSON dokumenty, ktoré vieme v reálnom čase upravovať a sú stále synchronizované.

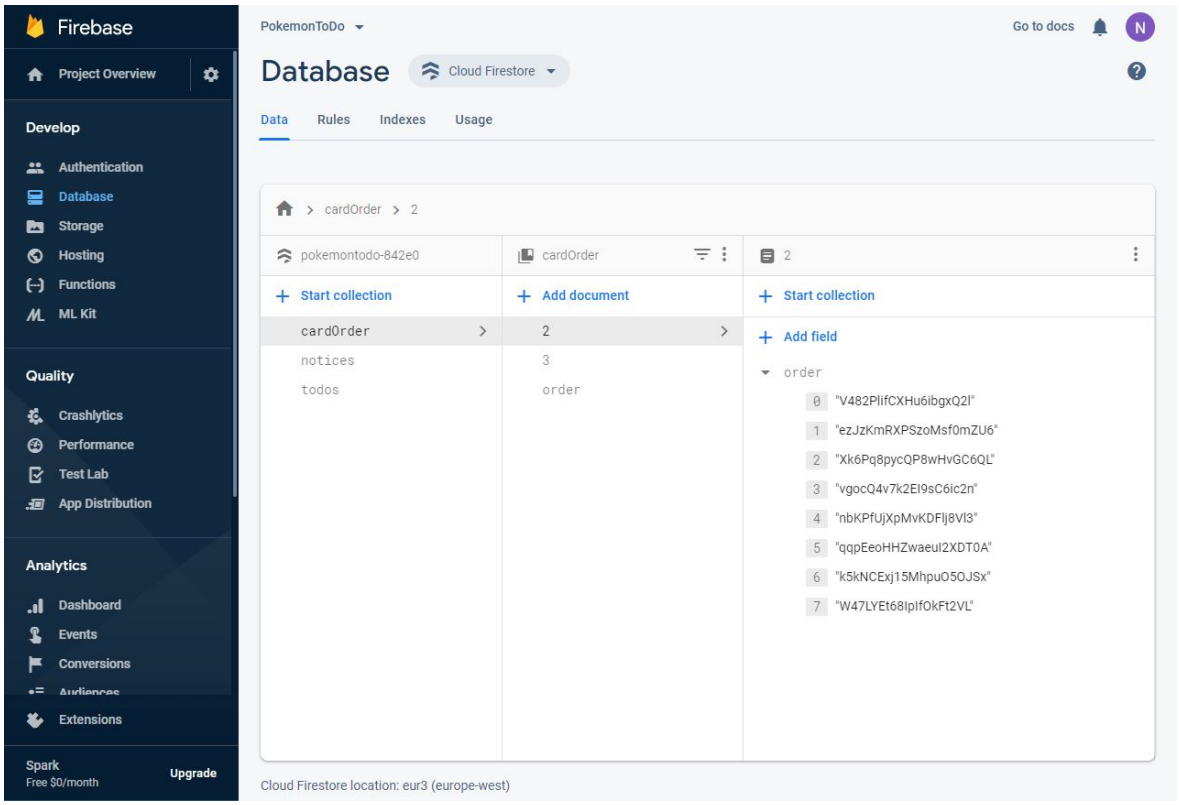
Medzi užitočné vlastnosti Firebase patrí možnosť autentifikácie pomocou vlastných účtov, no aj prepojením účtov tretích strán ako napríklad Github, Facebook, atď. To zároveň umožnuje prístup k dátam používateľa z týchto webových aplikácií, čo využívame aj v tejto práci pre získavanie dát z Githubu.

Taktiež ponúka možnosť ukladania dát vo Firebase Storage, v ktorom sa dajú ukladať rôzne typy dát. Pre mobilné aj webové aplikácie, ktoré to podporujú, ponúka možnosť zasielania notifikácií a taktiež možnosť Hostingu celej aplikácie. V prípade problémov s jednou z týchto služieb ponúka analýzu problémov a výkonu, čo uľahčuje hľadanie problémov a taktiež možnosť testovania.

Pre prácu so všetkými týmito nástrojmi má Firebase vytvorené knižnice. O autentifikáciu, storage a databázu sa v Reacte stará knižnica react-redux-firebase, ktorá vďaka jednoduchému api umožňuje prácu s tmito nástrojmi. Ďalšia užitočná knižnica je redux-firestore, ktorá má na starosti spravovanie stavu v Cloud Firestore, čo nám umožňuje využívať Firebase databázu aj ako stav v našej Reactovej aplikácii.

1.10 REST API

API, teda Application Programming Interface, je kód, ktorý umožňuje dvom softvérom medzi sebou komunikovať. To znamená, že vývojár vytvorí API na serveri, kde s ním klient môže komunikovať a využívať jeho služby.



Obr. 1.4: Webové užívateľské prostredie Firebase

Skratka REST označuje Representational State Transfer. To znamená, že určuje ako má API vyzerať a aké pravidlá má vývojár dodržiavať pri vytváraní svojho API.

1.11 Github

Git [9] je verziovací nástroj na zdieľanie zdrojových súborov softvérových projektov. Je založený na myšlienke decentralizovaných rovnocenných repozitárov.

Github je open-source webová služba určená na ukladanie a prístup ku repozitárom. Umožňuje ukladanie zdrojových kódov vo viacerých programovacích jazykoch a sleduje všetky zmeny v týchto súboroch. Keďže je Github využívaný na ukladanie rôznych typov textových súborov a ich verzií, využíva sa aj na zadávanie a riešenie úloh. Z tohto dôvodu naša webová aplikácia komunikuje s Githubom, získava a ukladá dáta. Na komunikáciu s aplikáciami využíva GraphQL API v4 a REST API v3, pričom v tejto práci využívame na komunikáciu REST API v3.

Pre uľahčenie práce s týmto API sme využili oficiálnu knižnicu Octokit, konkrétne Octokit/rest.js pre JavaScript. Táto knižnica pre nás zjednodušuje prácu s REST API, pretože žiadosti nemusíme posielať ako text, ale môžeme priamo využívať funkcie, ktorá táto knižnica ponúka.

Kapitola 2

Použité bakalárske práce

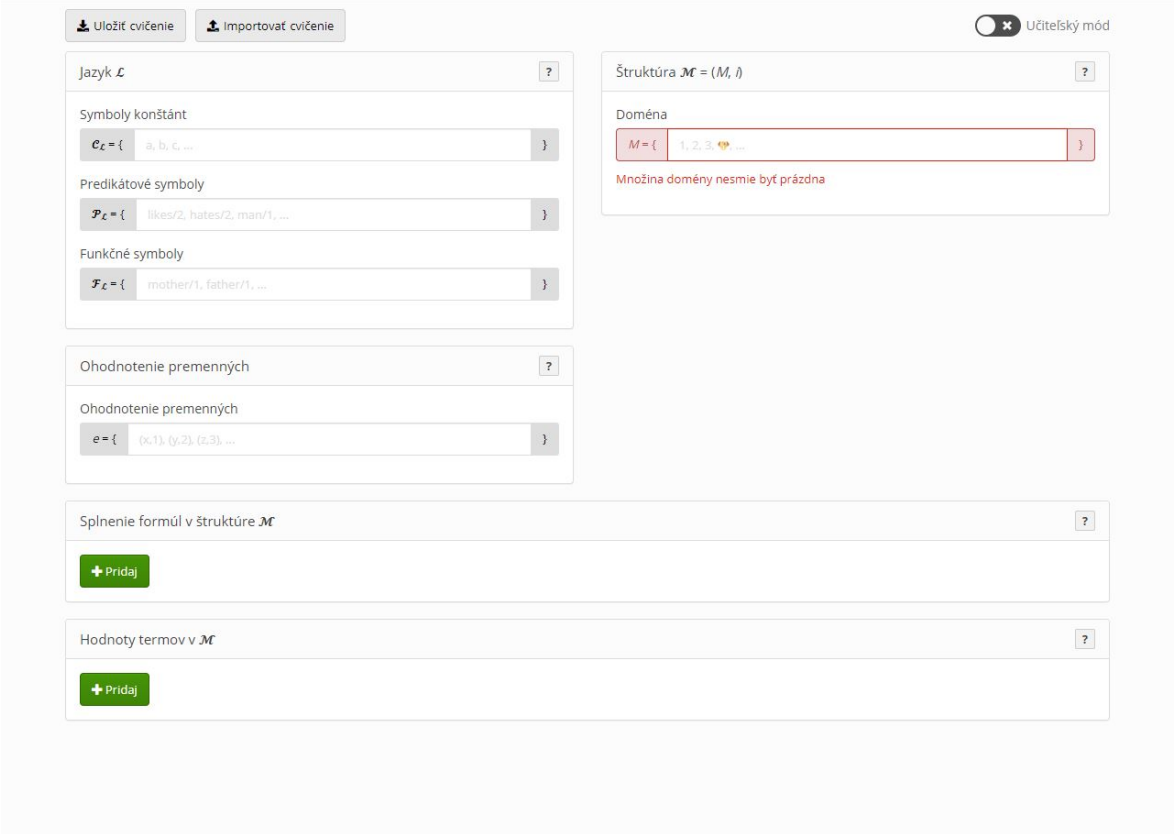
Cieľom interaktívneho pracovného hárku, ktorého vývoju sa venuje naša práca, je integrovať nástroje na podporu výučby logiky, ktoré vznikli v rámci predchádzajúcich bakalárskych prác. V tejto kapitole ich krátko predstavíme.

2.1 Prieskumník sémantiky logiky prvého rádu

Práca Milana Cifru [10] sa zaoberá tvorbou webovej aplikácie, ktorá ponúka študentom precvičovanie štruktúr logiky prvého rádu. V aplikácii si používateľ môže definovať ľubovoľnú konečnú štruktúru, jazyk a výrazy (formuly a termy). Výrazy aplikácia vyhodnocuje na základe definovanej štruktúry. Taktiež kontroluje syntax výrazov, čím vynucuje ich správny zápis. Aplikácia môže byť v dvoch módoch, ktoré si používateľ prepína - v študentskom a učiteľskom, ktorý umožňuje zamykanie výrazov a častí definície štruktúry. Vytvorenú štruktúru a výrazy je možné exportovať do textového súboru, a ten sa dá naspäť importovať. Aplikácia je naprogramovaná pomocou knižníc React a Redux.

2.2 Educational tools for first order logic

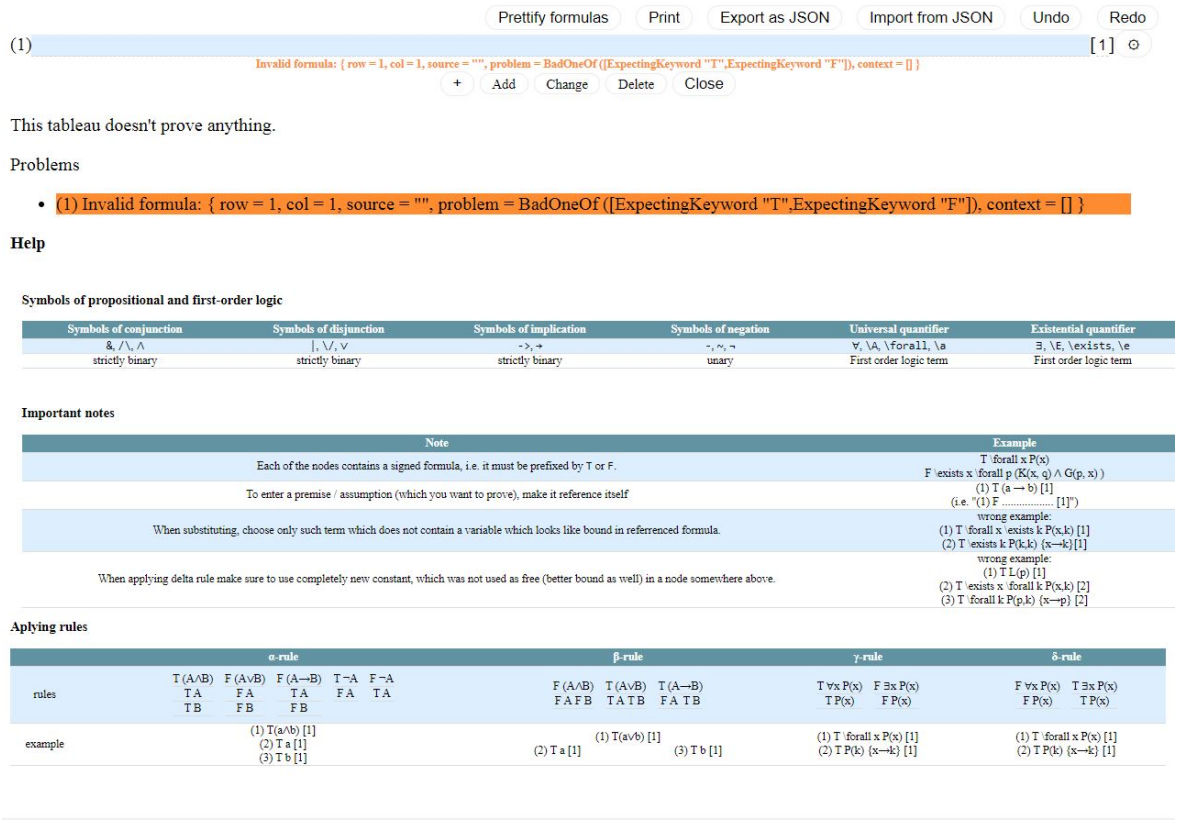
Aplikácia vytvorená Alexandrou Nyitraiovou [11] slúži na vytváranie dôkazu podľa metódy analytického tabla. Dôkaz nie je vytváraný automaticky, ale kontroluje každý krok užívateľa hneď po jeho vykonaní, v prípade omylu vypíše pri príslušnej formule chybu, týmto je aplikácia zameraná na edukáciu. Editor podporuje dôkazy v prvorádovej ale aj výrokovej logike. Dôkazy sú zobrazované ako stromy a každý vrchol je reprezentovaný formulou, tie sa odvodzujú pomocou základných štyroch pravidiel: alfa, beta, gama a delta. Aplikácia je naprogramovaná v jazyku Elm.



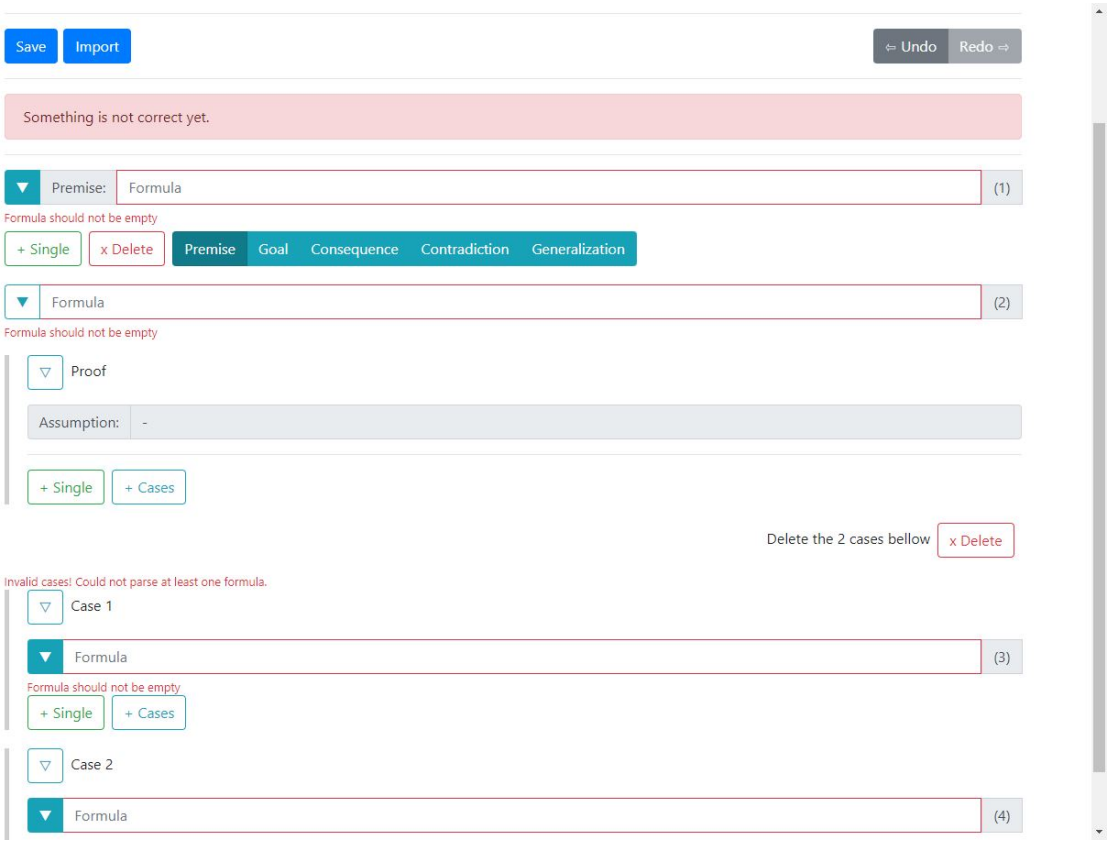
Obr. 2.1: Prieskumník sémantiky logiky prvého rádu

2.3 A proof assistant for first-order logic

Aplikácia Zoltána Onódyho [12] funguje ako interaktívny webový dokazovací asistent, ktorý podporuje tri typy dôkazov: priamy dôkaz, dôkaz analýzou prípadov a dôkaz sporom. Keďže je dôkaz založený na reťazení tvrdení, aplikácia overuje či sú nové tvrdenia dôsledkami tých predchádzajúcich a nejakého inferenčného pravidla. Používateľ do aplikácie zadáva sformalizované prvorádové formuly, ktoré sú potom vyhodnocované. Aplikácia je naprogramovaná v jazyku Elm.



Obr. 2.2: Educational tools for first order logic



Obr. 2.3: A proof assistant for first-order logic

Kapitola 3

Existujúce riešenia

Naša aplikácia má slúžiť na riešenie a zadávanie úloh rôznych typov pre študentov matematickej logiky. Na riešenie a zadávanie úloh už existujú aplikácie, ktoré majú podobnú funkčnosť. Medzi tieto aplikácie patrí napríklad Jupyter Notebook, MATLAB a Wolfram Mathematica. Tieto aplikácie fungujú na princípe Notebook interface, čiže výpočtového poznámkového bloku. To znamená, že slúžia ako virtuálny poznámkový blok používaný pre literálové programovanie. Je to programovacia paradigma, v ktorej program dostane vysvetlenie svojej logiky v prirodzenom jazyku s kúskami makier a tradičného kódu, z ktorého sa dá vygenerovať skompilovateľný zdrojový kód. [7] Keďže však tieto aplikácie nespĺňajú všetky naše požiadavky, slúžia ako príklad a inšpirácia ako sa v takomto interaktívnom prostredí dá pracovať a ako má vyzerať.

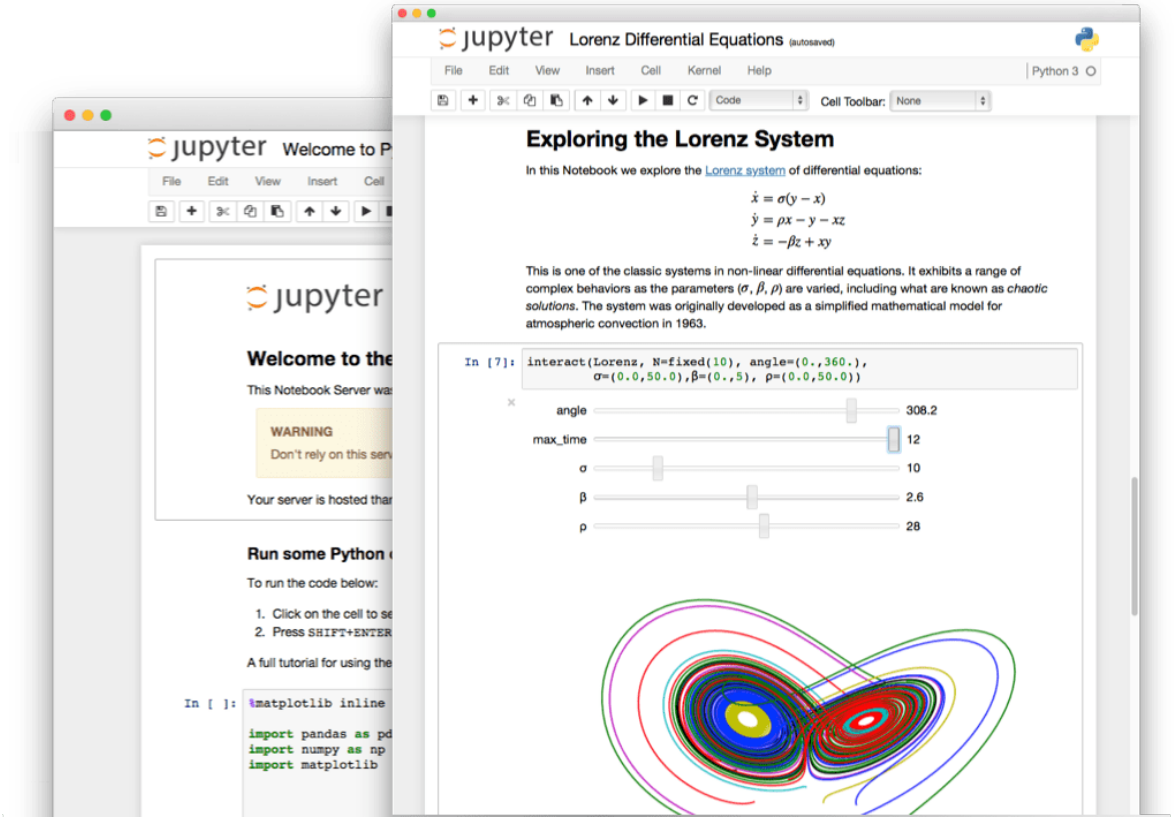
3.1 Jupyter Notebook

Jupyter Notebook [13] je interaktívne výpočtové prostredie vo webovom prehliadači. Prostredie Jupyter umožňuje pripojenie sa na výpočtové jadrá rôznych programovacích jazykov, je teda možné písať v ňom kúsky kódu v rôznych jazykoch do kódových buniek. Výsledky, ktoré pomocou tohto kódu získame, dokáže Jupyter zobraziť v rôznych formách - textovej, grafickej a tiež interaktívnej.

Slúži teda na vytváranie a zdieľanie Jupyter notebook dokumentov. Tieto dokumenty obsahujú JSON dokument a zoradený zoznam input/output políčok, ktoré môžu obsahovať kód, text (podporujúci Markdown), matematické výroky atď. Dokument tohto typu sa dá konvertovať do ďalších rôznych formátov ako HTML, LaTeX, PDF, Markdown, Python. Vďaka možnosti konvertovať dokument do HTML, je možné dokumenty prezerať aj priamo priamo v prehliadači.

Jupyter Notebook teda spĺňa takmer všetky podmienky, ktoré sme mali, keďže umožňuje zapisovať ako matematické výroky tak aj programovať v rôznych jazykoch. Pre efektívnu výuku nášho predmetu je však potrebné využívať spomenuté bakalár-

ske práce, ktoré sú v rôznych programovacích jazykoch – JavaScript a Elm. Jupyter Notebook však nepodporuje priame zakomponovanie týchto aplikácií, keďže tieto jazyky niesú priamo podporované. V prípade JavaScriptu by to ešte možné bolo, no pre Elm existuje bohužiaľ iba jeden rozrobený projekt na zabezpečenie funkčnosti Elmu v Jupyter prostredí a tento projekt je už niekoľko rokov neaktívny.



Obr. 3.1: Jupyter Notebook interface

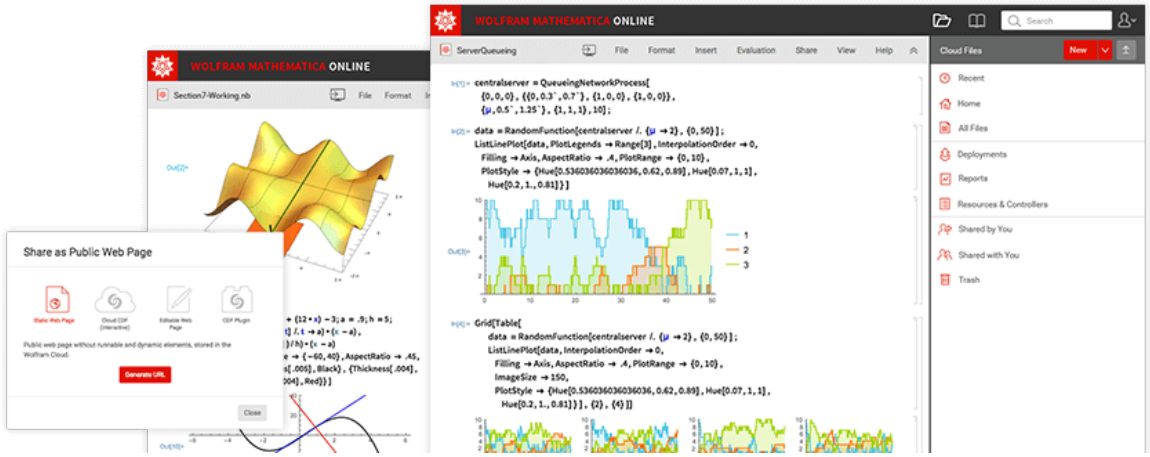
3.2 Wolfram Mathematica

Wolfram Mathematica je prvý systém, ktorý fungoval ako výpočtový poznámkový blok, ktorý sa postupne vyvíja. V súčasnosti ide o systém schopný výpočtov vo veľa vedeckých oblastiach. Delí sa na dve časti, kernel a front end. Kernel je program, ktorý je jadrom operačného systému počítača a stará sa o kontrolu celého systému. V tomto prípade pracuje vo vlastnom programovacom jazyku Wolfram Language a stará sa o vyhodnocovanie zadávaných výrazov a vracia výsledky, ktoré sa potom zobrazia.

Front end slúži na vytváranie a upravovanie Notebook dokumentov, ktoré obsahujú formátovaný text, časti kódu, rôzne grafické objekty, tabuľky atď. Obsah týchto dokumentov je rozdelený do buniek, ktoré sú hierarchicky usporiadané vďaka čomu sa dá dokument jasne rozdeliť. Obsahom týchto buniek môžu byť rôzne vstupy vo forme

príkazov a funkcií, v prípade výstupov sa tu zobrazujú texty, grafické prvky, atď.

Mathematica ponúka spoluprácu s rôznymi programovacími jazykmi a technológiami, čo zabezpečuje široké využitie pre výpočty, modelovanie, spracovanie obrazu atď.

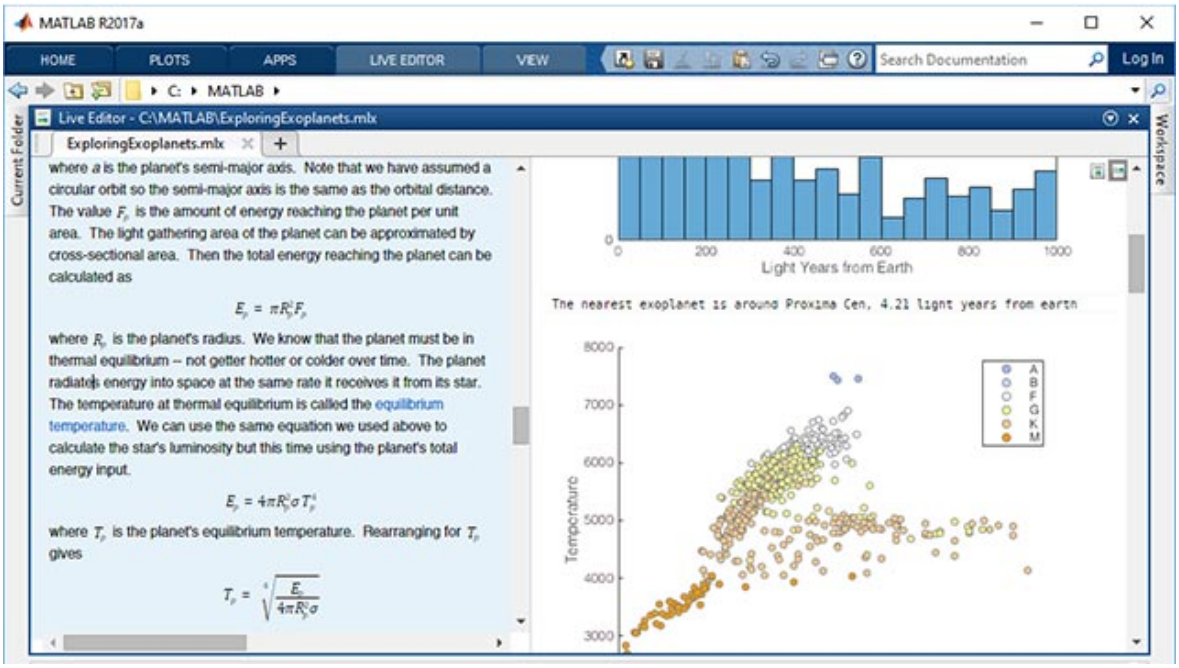


Obr. 3.2: Wolfram Mathematica Interface

3.3 MATLAB

MATLAB [8] je interaktívne programové prostredie a skriptovací programovací jazyk. Ponúka veľa funkcií ako vykresľovanie 2D a 3D grafov, implementáciu algoritmov, prezentáciu dát, vytváranie používateľských rozhraní a podobne. Názov MATLAB vznikol skrátením slov Matrix Laboratory, čiže Maticové Laboratórium, pretože kľúčovou dátovou štruktúrou pri výpočtoch sú práve matice.

MATLAB je silným nástrojom, ktorý zvládne aj náročné výpočty a simulácie, nič z toho však nepotrebujeme. Keďže cieľom tejto práce je uľahčiť študentom aj vyučujúcim prácu na predmete a riešenie úloh, MATLAB je na tento účel až príliš komplikovaný a keďže na jeho používanie je potrebná licencia, tak aj finančne náročný.



Obr. 3.3: Matlab interface

Kapitola 4

Ciele práce

V tejto kapitole rozoberieme účel a ciele, ktoré má naša aplikácia spĺňať. V prvej časti budeme hovoriť o tom aký účel a pre koho má naša aplikácia. Ďalej budeme hovoriť a požiadavkách na našu aplikáciu, ktoré chceme splniť aby vyhovovala všetkým jej používateľom.

4.1 Účel aplikácie

Aplikácia, ktorou sa zaoberá naša bakalárska práca má zjednodušiť výuku matematickej logiky pre študentov aj učiteľov. V súčasnosti prebieha výučba teoretickej časti predmetu, pre ktorý je táto aplikácia určená primárne v papierovej forme, čo zťažuje prácu študentom aj vyučujúcim. Existuje niekoľko minulých bakalárskych prác, ktoré pomáhajú študentom riešiť rôzne typy úloh, no nenachádzajú sa prehľadne na jednom mieste a ich výstupy niesú priamo dostupné vyučujúcim pre ich kontrolu a hodnotenie.

Cieľom práce je teda vytvoriť aplikáciu, ktorá bude fungovať ako interaktívny pracovný hárok, v ktorom budú vyučujúci schopní zadávať nové úlohy, prípadne upravovať už existujúce zadania. Študenti budú potom môcť tieto úlohy vypracovávať a odovzdávať bez nutnosti využívania papierovej formy.

4.2 Požiadavky na funkčnosť aplikácie

V tejto časti budeme hovoriť o funkciách, ktoré má naša aplikácia mať, aby bola plne využiteľná počas výučby. Základné požiadavky boli určené pri zadávaní práce, no iné vznikali postupne počas konzultovania so školiteľom práce.

4.2.1 Používateľ

- Po načítaní stránky môžu nastať 2 prípady

1. Používateľ nie je prihlásený, zobrazí sa mu prázdna stránka so žiadosťou o prihlásnie a možno prihlásenia v navigácii.
2. Používateľ je prihlásený, zobrazí sa mu celý obsah stránky a možnosť odhlásenia.

- Prihlásenie používateľa prebieha cez jeho Github účet, čo nám po používateľovom súhlase umožní získať prístup k jeho repozitárom.
- Pokiaľ sa používateľ neodhlási, zostáva na stránke prihlásený aj po jej zavretí.

4.2.2 Prehliadač repozitárov

- Aplikácia po spustení načíta používateľove dáta z Githubu a spustí prehliadač repozitárov. Ten sa môže dostať do 4 stavov:
  1. Nie je vybraný žiadny repozitár, používateľovi sa teda zobrazí iba jedno tlačidlo, po kliknutí na toto tlačidlo sa používateľovi zobrazí rozbaľovací zoznam všetkých jeho repozitárov, ktoré boli načítané po jeho prihlásení.
  2. Po výbere repozitára aplikácia získa zoznam vetiev repozitára a objaví sa druhé tlačidlo pre výber vetvy.
  3. Výberom vetvy sa načítajú súbory, ktoré daný repozitár vo vybranej vetve obsahuje. Vytvorí sa omrvinková navigácia pre ľahšiu orientáciu v repozitári. Zároveň sa vytvorí zoznam súborov a zložiek, na ktoré bude možné kliknúť. Kliknutím na zložku sa otvorí a pridá sa do omrvinkovej navigácie.
  4. Kliknutím na súbor sa tento súbor otvorí a jeho obsah, ak to bude možné sa načíta. V prípade že pôjde o zadanie úlohy, otvorí sa prehliadač zadaní. V inom prípade sa obsah súboru zobrazí pod zoznamom všetkých súborov.
- Aplikácia bude schopná ovládať prehliadač repozitárov aj podľa URL adresy.
- Po načítaní súboru so zadaniami sa prehliadač zavrie a ostanú iba tlačidlá na výber repozitára, vetvy a omrvinková navigácia. Interakciou s týmito prvkami sa prehliadač znova otvorí.

4.2.3 Prehliadač zadaní

- Pre vybrané zadanie sa zobrazí zoznam buniek, ktoré obsahujú zadanie úlohy, priestor na riešenie alebo jednu z edukačných aplikácií
- Bunky sa dajú pridávať aj mazať. Pridávanie bunky bude pomocou tlačidla, ktoré sa bude nachádzať po každou bunkou a zobrazí sa iba po nadídení kurzorom na

tento priestor. Po kliknutí sa vytvorí nová prázdna bunka za bunkou, na ktorej tlačidlo sme klikli.

- Bunky budú podporovať Markdown, text napísaný v Markdowne sa vykreslí ako text so všetkými zadanými vlastnosťami.

- Bunka sa bude dať otvoriť, pričom po nadídení kurzorom na bunku sa zvýrazní. Po dokončení úprav sa bude dať uložiť a opäť zatvoriť. Každá bunka bude teda mať 2 stavy:

    1. Bunka nieje otvorená, zobrazuje iba vyrenderovaný text a zaberá menej miesta. Po kliknutí na bunku sa táto bunka otvorí a prejde do druhého stavu.
    2. Bunka je otvorená, zobrazuje sa nám editor pre Markdown alebo jedna z edukačných aplikácií a tlačidlo na uloženie dát. Po dokončení jej úprav klikneme na tlačidlo Uložiť, ktoré uloží nové dáta a zavrie bunku, ktorá sa vráti do prvého stavu.

- Bunky sa dajú presúvať, kliknutím a podržaním kurzora vieme s bunkou hýbať a meniť jej poradie, počas pohybu sa ostatné bunky presúvajú a po pustení sa bunka vloží na vybrané miesto.

Kapitola 5

Návrh

V tejto kapitole sa budeme venovať čo najpresnejšiemu návrhu našej aplikácie. Pri jeho vytváraní sme sa inšpirovali už existujúcimi riešeniami.

Výsledkom návrhu je webová aplikácia, ktorá slúži na zlepšenie výučby matematickej logiky pre študentov aj vyučujúcich. Ako implementačný jazyk sme zvolili JavaScript, konkrétne knižnicu React, ktorá nám umožňuje vytvárať samostatné komponenty, využiť riešiče ako vlastné komponenty a rýchlu prácu s úložiskom. Na ukladanie dočasných dát aplikácie využijeme knižnicu React-Redux. Aplikácia je pripravená na ďalšie rozšírenia prípadne možnosť použiť ju na iný účel len s malými úpravami.

5.1 Používateľské prostredie

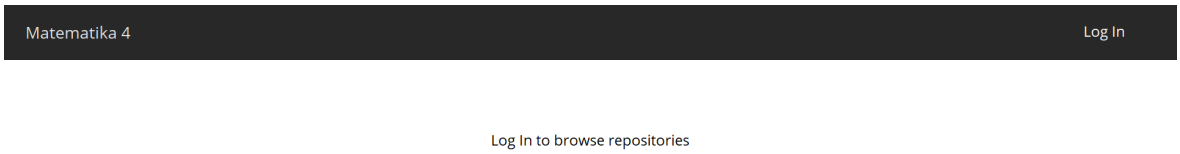
Keďže nechceme aby bola aplikácia mätúca a príliš zložitá, zvolili sme jednoduchý a intuitívny dizajn používateľského prostredia, ktoré nebude rušiť a rozptyľovať nepodstatnými dizajnovými prvkami. Aplikácia je zameraná primárne pre zobrazenie na veľkej obrazovke, na vďaka Bootstrapu je využiteľná aj na mobilných zariadeniach, kde sa prvky automaticky prispôsobia.

5.1.1 Neprihlásený používateľ

Neprihlásený používateľ nemá k dispozícii takmer žiadnu časť aplikácie, po jej spustení sa mu zobrazí prázdna stránka so žiadosťou o prihlásenie. Prihlásiť sa môže v pravom hornom rohu kliknutím na nápis Log In.

5.1.2 Prihlásený používateľ

Používateľ po prihlásení získa prístup k plnej funkcionalite stránky, každý používateľ má rovnaké práva na prácu s dátami. Vo vrchnej časti obrazovky sa zobrazí navigácia, ktorá obsahuje iba názov stránky a tlačidlo na odhlásenie. Pod navigáciou sa nachádza



Obr. 5.1: Stránka neprihláseného používateľa

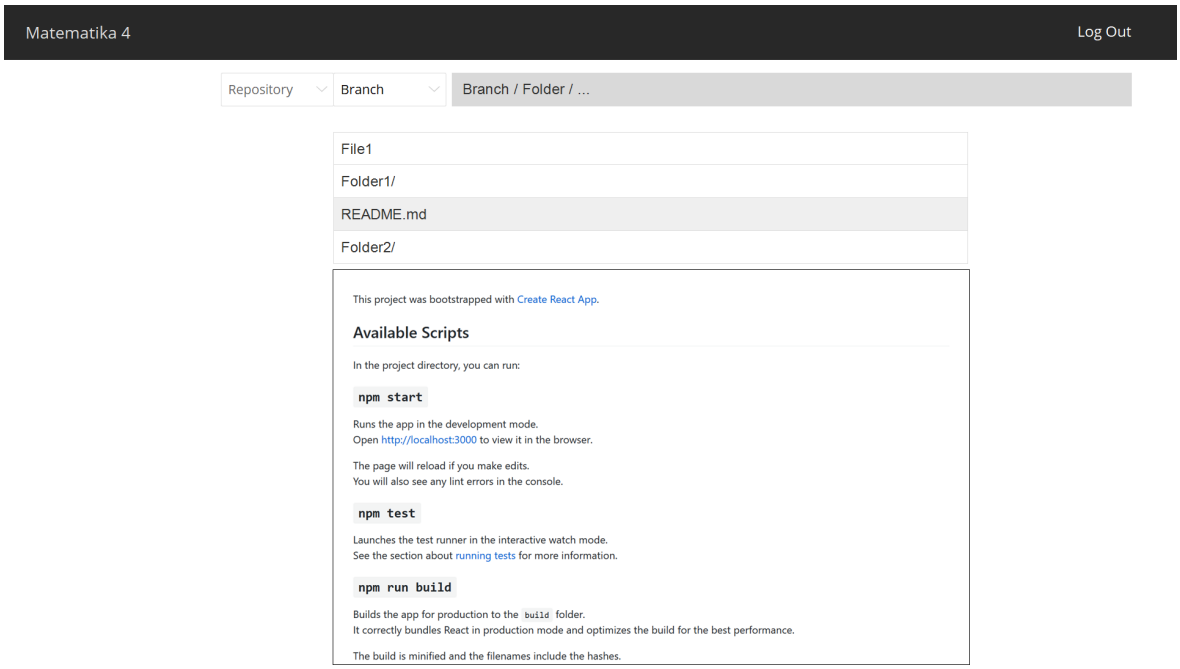
prehliadač repozitárov, v ktorom si používateľ dokáže prechádzať všetky svoje verejné aj súkromné repozitáre.

Na vrchu prehliadača sa nachádza tlačidlo pre výber konkrétneho repozitára, vedľa neho tlačidlo na výber vetvy v tomto repozitári a lišta omrvinkovej navigácie, ktorej prvým prvkom bude aktuálne zvolená vetva. Omrvinková navigácia bude slúžiť pre návrat do predchádzajúcich zložiek repozitára. Ten sa zobrazí ako zoznam súborov a zložiek, pričom zložky budú jasne odlíšené pridaním lomítka na koniec názvu. Kliknutím na takúto zložku sa pridá jej názov do omrvinkovej navigácie a v zozname sa načítajú nové súbory a zložky. Kliknutím na súbor, ktorý neobsahuje zadania úloh sa zobrazí jeho obsah pod prehliadačom.

5.1.3 Prehliadanie zadaní

Po otvorení súboru v prehliadači repozitárov, ktorý obsahuje zadania úloh sa tieto zadania načítajú ako samostatné karty, prehliadač sa vypne a ostane z neho viditeľné iba tlačidlo na výber repozitára a omrvinková navigácia. Kliknutím na ňu alebo výberom iného repozitára sa prehliadač repozitárov znova otvorí a súbor so zadaniami sa zatvorí.

Každá karta je najskôr zbalená a obsahuje vykreslený text, ktorý je napísaný v Markdowne. Karty sa dajú presúvať a pod každou z nich sa nachádza tlačidlo na vytvorenie novej prázdnej karty. Táto karta sa zobrazí pod kartou, ku ktorej patrilo tlačidlo.



Obr. 5.2: Stránka prihláseného používateľa

5.1.4 Práca na zadaní

Každá karta sa dá otvoriť, čím sa spustí editor a je možné ju upravovať. V pravom hornom rohu karty sa nachádza rozklikávacie tlačidlo, v ktorom bude možnosť vymazať kartu a v pravom dolnom rohu tlačidlo pre uloženie. Pre naše účely budú existovať 2 druhy kariet:

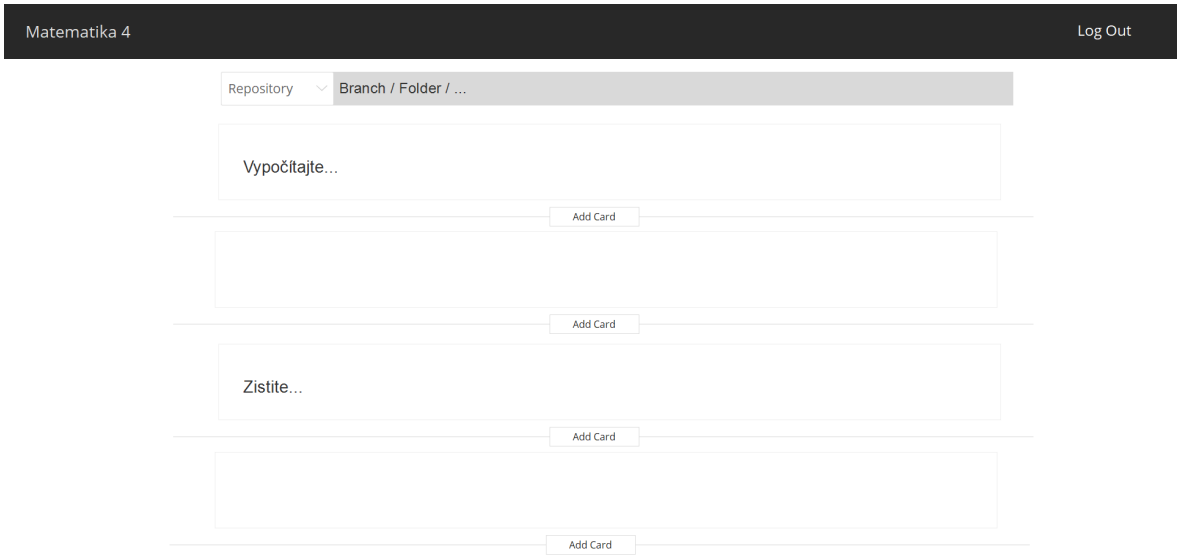
- Textová karta, ktorá po otvorení zobrazí Markdown editor, v ktorom bude možné písať a zároveň zobrazovať upravený text.
- Karta s riešičom, v ktorom sa bude dať pracovať s jedným z implementovaných riešičov.

5.2 Ukladanie dát

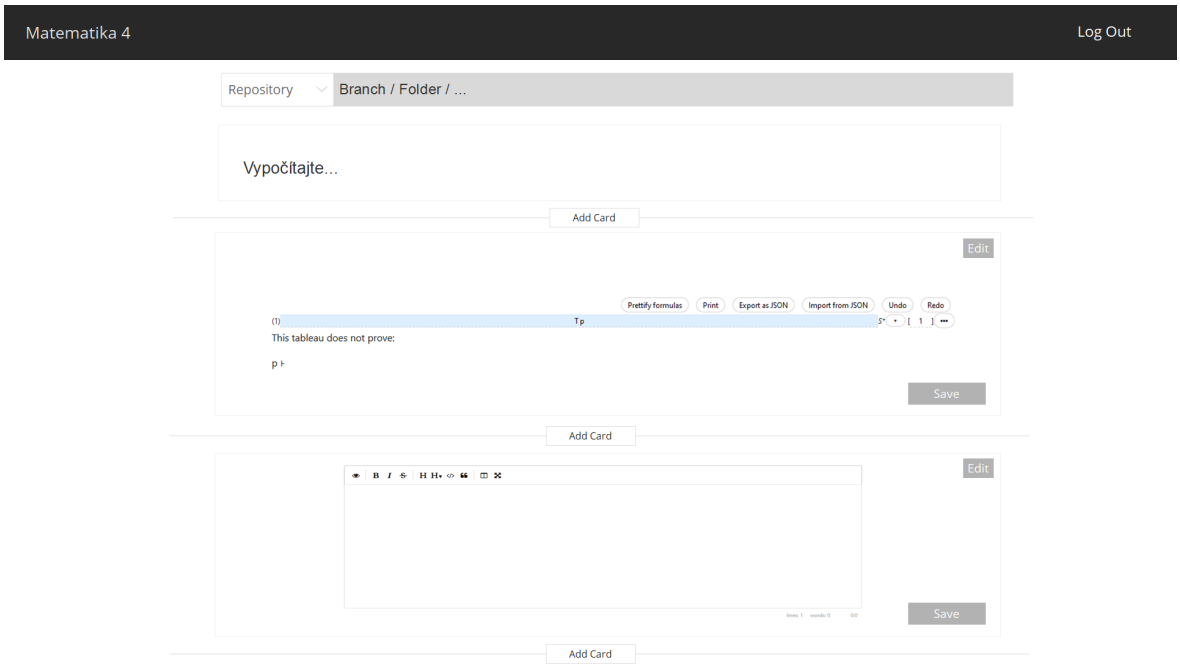
V aplikácii bude prebiehať veľa práce s dátami, ktoré je potrebné niekde uchovávať. Nie všetky dáta však potrebujeme uchovávať rovnaký čas a pristupovať k nim v rovnakých intervaloch a rovnakou rýchlosťou. Máme teda 3 typy dát, ktoré vieme rozdeliť podľa ich životnosti.

5.2.1 Krátka životnosť

Aplikácia pracuje s veľa dátami, ktoré sa často menia a nieje potrebné ich trvalo uchovávať, preto majú najkratšiu životnosť. Tieto dáta sa získavajú za behu aplikácie a



Obr. 5.3: Zoznam zadaní



Obr. 5.4: Práca so zadaním

niesú potrebné pri jej ďalšom spustení a strácajú sa po zatvorení aplikácie. S krátkou životnosťou sú teda dáta aktuálneho stavu našej aplikácie rôznych typov, ku ktorým potrebujeme rýchly prístup a vykonávať veľa zmien.

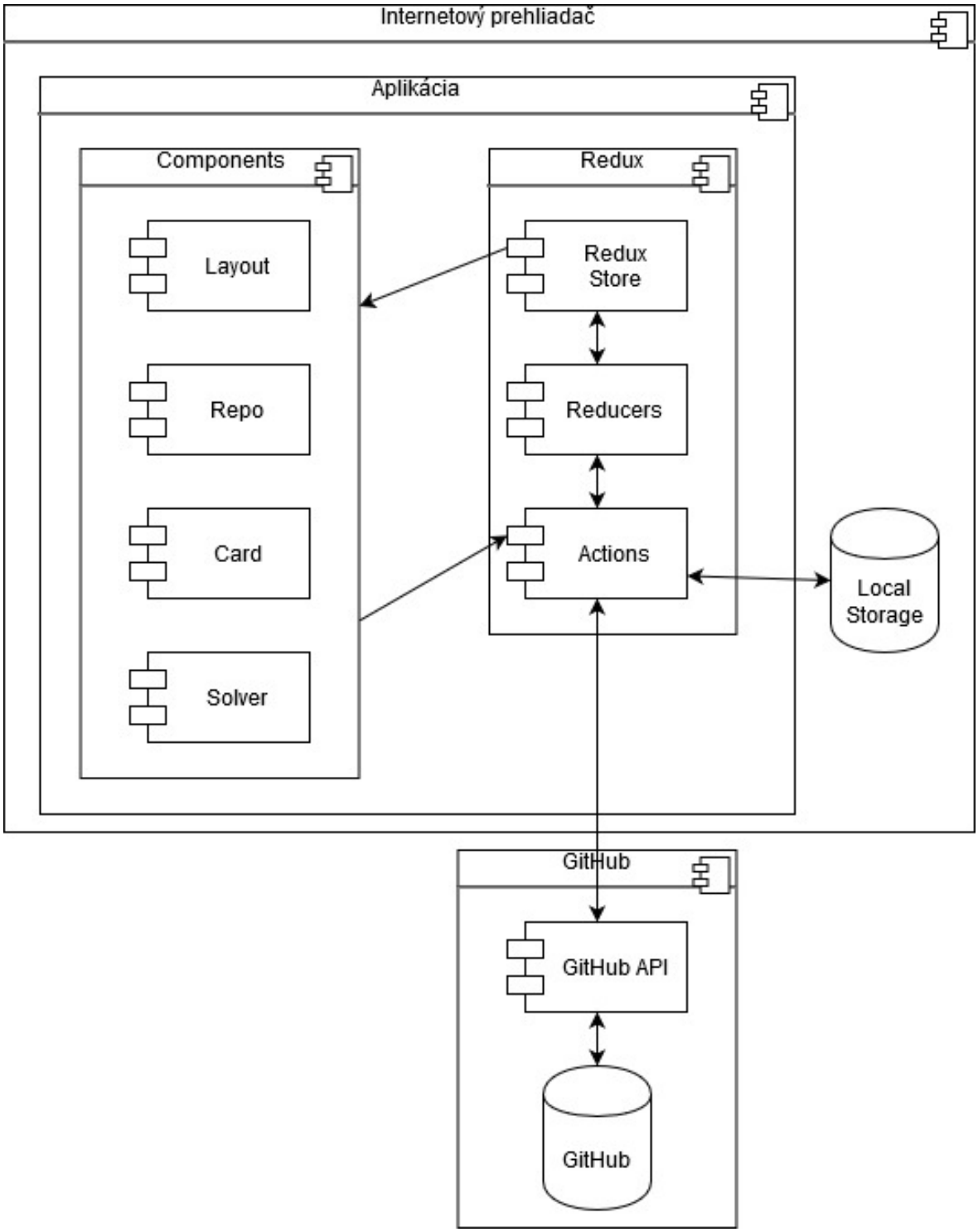
5.2.2 Stredná životnosť

Chceme taktiež aby bola naša aplikácia pre používateľov príjemná a zjednodušovala im prácu. Potrebujeme teda aby sa používateľ nemusel po každom zapnutí aplikácie znova prihlasovať, ale bolo jeho prihlásenie automatické, až kým sa sám neodhlási. Keďže naša aplikácia umožňuje prezeranie repozitárov a súborov v nich uložených, taktiež chceme aby sa používateľ po obnovení stránky dostal v prehliadači repozitárov na miesto, kde sa nachádzal predtým. Na ukladanie týchto dát už krátkodobé úložisko nevyhovuje, pretože si potrebujeme niektoré dáta uchovávať aj po obnovení aplikácie. Tieto dáta budeme uchovávať ako znakové reťazce v úložisku so strednou životnosťou.

5.2.3 Dlhá životnosť

Hlavným účelom našej aplikácie je vytvoriť prostredie pre zadávanie a riešenie úloh. Preto potrebujeme persistentné úložisko, do ktorého sa budú môcť dlhodobo ukladať potrebné dáta študentov a aby k nim mali používatelia prístup aj mimo našej aplikácie. Zadania, ktoré budú študenti vypracovávať sa uchovávajú priamo v krátkodobom úložisku aplikácie, po stlačení tlačidla na uloženie sa tieto dáta pripravia a odošlú do dlhodobého úložiska.

Pre tento typ dát sme pôvodne chceli využiť databázu, ktorú nám ponúka Firebase. Tá však našej aplikácii úplne nevyhovovala, pretože by sme museli obmedziť počet čítaní a zápisov do databázy, jej štruktúra by bola zložitejšia a priamy prístup k uloženým dátam by bol náročnejší. Preto sme sa rozhodli využiť GitHub, ktorý ponúka možnosť dlhodobo ukladať súbory, porovnávanie ich verzií a predmet Matematika 4 už aktívne využíva GitHub na zadávanie a kontrolu praktických úloh. Pre odosielanie dát z našej aplikácie využijeme GitHub API, ktoré nám umožní priamo komunikovať s GitHubom a ukladať v ňom dáta. Súbory, ktoré budeme ukladať budú typu JSON, tento typ súborov umožňuje jednoduchú konverziu dát na objekt a naopak. Riešiče, ktoré naša aplikácia využíva taktiež využívajú ako vstupy aj výstupy súbory typu JSON, vďaka čomu ich môžeme priamo ukladať a v prípade konfliktov medzi verziami je jednoduché tieto konflikty vyriešiť.



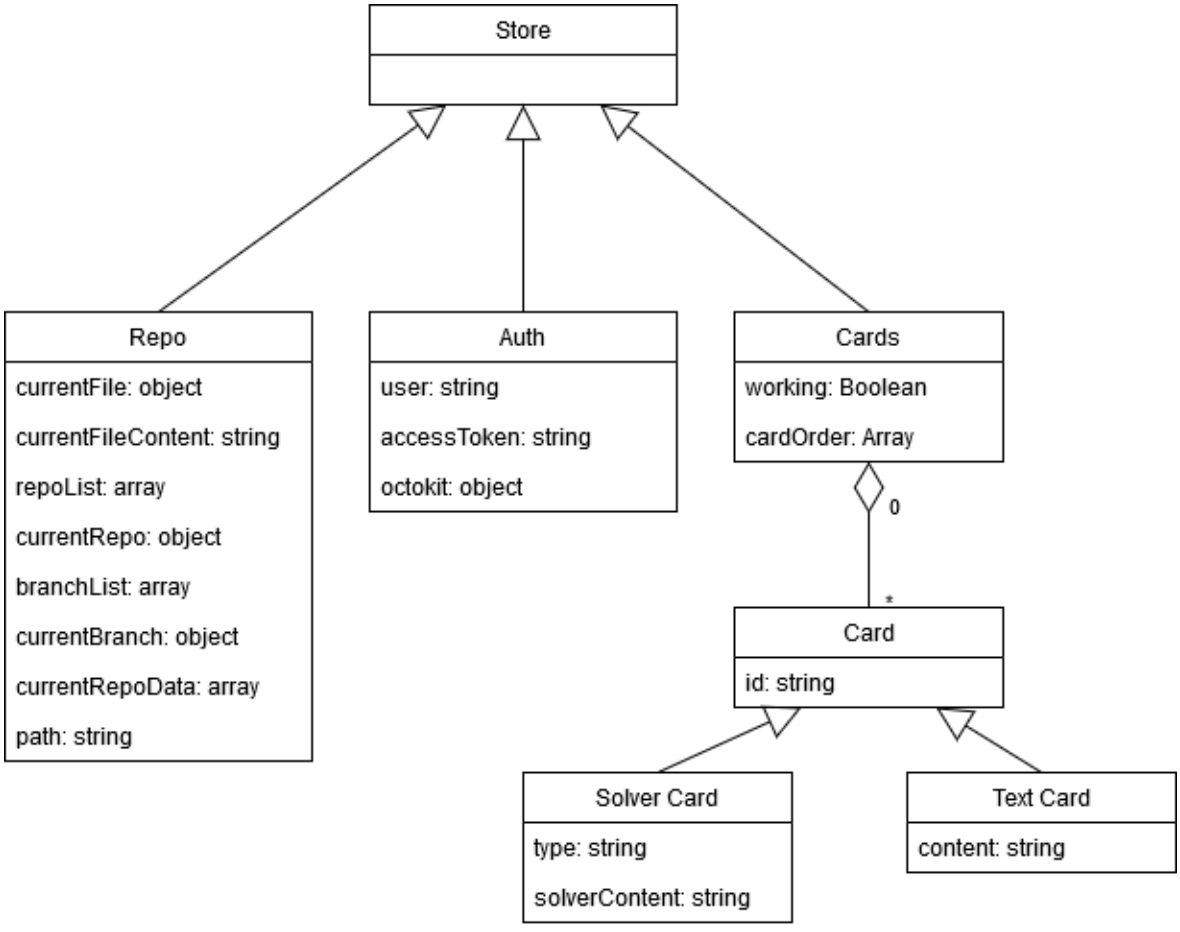
Obr. 5.5: Architektúra aplikácie pre prácu s dátami

5.3 Štruktúra aplikácie

Aplikácie je rozdelená do niekoľkých zložiek, podľa zamerania jej komponentov. Pre ukladanie dát s krátkou životnosťou využijeme knižnicu Redux, ktorá s Reactom dobre spolupracuje a umožňuje nám rýchlu prácu s dátami a vykreslovanie zmien. Pre uloženie aktuálnych dát prehliadača repozitárov, využijeme stav Repo a informácie o používateľovi budú uložené v Auth.

Zadania sú uložené v Cards, kde sa ukladá informácia o tom, či práve niektoré zadanie komunikuje s dlhodobým úložiskom, poradie jednotlivých zadaní v zozname a samotné zadania. Každé zadanie je jedného z 2 typov.

Prvým typom zadaní sú textové zadania, pri ktorých sa uchováva jeho textový obsah. Druhým typom sú zadania obsahujúce riešiče, ktoré si uchovávajú o aký typ riešiča ide a aké dáta sú preňho uložené.



Obr. 5.6: Diagram uloženia dát v stave Redux Store

5.4 Riešiče

V tejto práci integrujeme riešiče vytvorené v rôznych rôznych jazykoch a zároveň chceme aplikáciu pripraviť na to, aby do nej bolo možné pridávať ďalšie takéto riešiče bez veľkých zásahov do existujúceho kódu. Pre implementáciu riešičov môžeme využiť 2 rôzne prístupy:

- Využiť iFrame, čiže vložiť celú samostatnú aplikáciu do našej. V tomto prípade je náročnejšie komunikovať s aplikáciu, ktorá sa nachádza v iFrame a v našom prípade potrebujeme dáta vkladať aj získavať.
- Pridať riešič ako komponent do našej aplikácie, pretože sú komponentového typu, pracujú samostatne a máme prístup ku ich kódu v prípade, že potrebujeme vykonať zmenu.

Keďže pri iFrame je komunikácia medzi naším a vloženým komponentom náročnejšia, zvolili sme druhý spôsob - vložiť riešič ako komponent, ktorému budeme vedieť priamo poslať vstupné dáta a meniť dáta v našom stave, ktoré budú následne ukladané. Pre vloženie riešičov naprogramovaných v jazyku Elm, využijeme existujúce knižnice na implementáciu Elmového kódu do Reactu. Pri vkladaní riešičov, ktoré sú naprogramované pomocou Reactu si vytvoríme vlastný komponent, v ktorom budeme vytvárať vkladanú aplikáciu ako celok s vlastným stavom, pričom k nim pridáme vlastný middleware, ktorý bude odchytávať dáta a ukladať ich v našom stave.

Kapitola 6

Implementácia

V tejto kapitole bližšie opíšeme niektoré z dôležitých a náročnejších častí aplikácie s ukážkami ich kódu. Pre zlepšenie prehľadnosti a zlepšenie funkčnosti využívame lokálny stav v Reduxe, do ktorého posielame dáta pomocou špeciálnych funkcií - Actions, v ktorých dáta spracujeme a potom ich odošleme do lokálneho stavu.

6.1 Štruktúra aplikácie

- V priečinku Components sa nachádzajú podpriečinky s prezentačnými komponentmi, ktoré sú v našej aplikácii využívané.
  - V priečinku Card sa nachádzajú prezentačné komponenty, ktoré slúžia na prácu so zadaniami.
  - Layout obsahuje komponent, ktorý slúži na zobrazenie a prácu s navigáciou.
  - Repo obsahuje komponenty, ktorých úlohou je zobraziť a pracovať s prehliadačom repozitárov.
  - Solvers má za úlohu pracovať s riešičmi úloh.
- Priečinok Config obsahuje iba jeden súbor, v ktorom sú všetky potrebné informácie pre prepojenie aplikácie so službami, ktoré ponúka Firebase
- V priečinku Constants sa nachádza iba súbor, v ktorom si definujeme globálnu premennú, ktorú budeme neskôr využívať pre Drag and Drop pri posúvaní naších kariet v zadaní.
- Všetky komponenty, ktoré súvisia s knižnicou Redux, sú uložené v priečinku Store.
  - Actions obsahuje akcie, ktoré jednotlivé komponenty spúšťajú kvôli práci s dátami v Reduxe a GitHube.

– Reducers má na starosti Reduxové reducery, ktoré majú za úlohu odchytávať akcie a upravovať dáta v Store podľa týchto dát.

6.2 Práca s dátami

6.3 Tok dát

V aplikácii bude prebiehať veľa práce s dátami v rámci aplikácie aj s vonkajším serverom. Pre uchovávanie dát využívame 3 typy úložiska, pričom každé má iný význam, ukladajú sa v nich rozdielne dáta a taktiež sú schopné udržiavať dáta po rozličnú dobu.

6.3.1 Redux

Pre správu a ukladanie dát, s ktorými naša aplikácia narabá najčastejšie využívame knižnicu Redux, ktorá nám umoňžuje vytvoriť globálne úložisko dát s názvom Store. V tomto úložisku sa nachádzajú všetky dáta potrebné pre správne fungovanie aplikácie, preto je potrebné aby k nim mali komponenty jednoduchý prístup. V tomto úložisku si uchovávame dáta pre prehliadač repozitárov, rozpracované zadania úloh a riešičov.

Aby sme zabezpečili prehľadnosť a čo najmenšiu závislosť medzi komponentmi, každý z komponentov samostatne napájame na toto úložisko, čím každý z naších komponentov získa prístup ku dátam. Druhá možnosť by bola pripojiť iba rodičovský komponent a potomkom poslať potrebné dáta, tým sa však stávajú viac prepojené a kód sa stáva neprehľadnejším a náročnejším na úpravy. Komponent je nepretržite pripojený na Store, vďaka čomu sa po zmene jeho stavu každý pripojený komponent v prípade potreby prekreslí, čo využijeme pri prihlásení a odhlásení používateľa, práci s prehliadačom repozitárov, zadaniami a riešičimi.

Komponenty potrebujú taktiež možnosť odosielať dáta do Store, ktorý však slúži ako stav a nemali by sme ho priamo meniť a preto využívame funkcie v Actions a Reducers. Actions sú funkcie, ktoré slúžia na odosielanie dát do Store. Zároveň nám umožňujú využívať rovnaké funkcie vo viacerých komponentoch bez duplikácie. Funkcie, ktoré náš komponent bude využívať k nemu opäť pripojíme a môžeme s nimi pracovať. V akciách môžeme vykonávať rôzne úpravy dát, získavať nové dáta atď. Vďaka tomu sprehľadníme náš komponent a iba vyvoláme požadované funkcie a potrebné úpravy prebehnú v nich. Akcie po dokončení všetkých potrebných úprav odošlú nové dáta Reducerom, tie odchytia poslané dáta a uložia ich do nášho Store.

6.3.2 LocalStorage

LocalStorage je priestor pre uloženie dát, ktorý nám ponúka webový prehliadač. Tieto dáta sú teda udržiavané dlhšie než tie v Redux Store a môžeme ich využívať opakovane. V našej aplikácii využívame LocalStorage pre ukladanie dát potrebných pre automatické prihlásenie používateľa po spustení aplikácie. Po prihlásení používateľa využijeme akciu, ktorá nám uloží používateľove meno a accessToken do tohto LocalStorage. Tie sú v ňom uchované pokiaľ sa používateľ neodhlási. Pri zapnutí aplikácie skontrolujeme, či máme v LocalStorage uložené dáta a ak áno, tak používateľa prihlásime.

6.3.3 GitHub

Naša aplikácia slúži na zadávanie a riešenie úloh, tie potrebujeme ukladať dlhodobo a musia byť dostupné aj mimo našej aplikácie. Preto na ukladanie naších zadaní využívame GitHub a jeho REST API. Ak komponent potrebuje získať alebo uložiť dáta, nepristupuje k nim priamo ale zavolá akciu, ktorá podľa zadaných údajov získa dáta z GitHubu pomocou funkcií z knižnice Octokit, počká na odpoveď a dáta následne uloží do Redux Store. Keďže je komponent pripojený na tento Store, zistí zmenu a získa dáta, ktoré si vyžiadal. Počas ukladania sa dáta komponentu väčšinou nezmenia, no odpoveď, ktorú akcia dostane uloží aplikácia do Store.

6.4 Prihlásenie a odhlásenie používateľa

Pre zobrazenie repozitárov a úloh musí byť používateľ prihlásený, inak je väčšina funkcionality aplikácie vypnutá a čaká na prihlásenie. Pre každého používateľa potrebujeme prístup k jeho repozitárom, ktoré má uložené na Gihube. To je jeden z dôvodov, pre ktoré sme sa rozhodli zakomponovať BaaS službu Firebase do našej aplikácie.

Táto služba ponúka jednoduché spravovanie používateľov vo svojej databáze. Zároveň však umožňuje využiť prihlásenie pomocou iných služieb medzi ktoré patrí aj Github, z ktorého používateľove dáta potrebujeme získavať. Kvôli bezpečnosti GitHub využíva systém OAuth, v ktorom sme museli najskôr našu aplikáciu zaregistrovať. Na prihlásenie využívame funkciu, ktorá je súčasťou balíka Firebase SDK. Keďže však potrebujeme možnosť prehliadať aj súkromné repozitáre, pridávame do hlavičky žiadosti rozsah prístupu řepo.

Počas prvého prihlasovania do našej aplikácie, sa používateľovi otvorí nové okno, v ktorom sa od neho žiada prihlásenie do služby Github a udelenie povolenia našej aplikácii pre prístup do jeho repozitárov. Po vyplnení údajov a potvrdení žiadosti sa táto požiadavka odošle a aplikácia čaká na odpoveď. V prípade, že je všetko správne používateľ je prihlásený a v odpovedi, ktorú aplikácia dostala sa nachádza accessToken, teda unikátny kľúč, vďaka ktorému získavame prístup na čítanie a zápis do

používateľových súborov. Tento accessToken spolu s menom používateľa ukladáme do localStorage prehliadača, čo je malé úložisko v prehliadači, do ktorého môžu aplikácie ukladať dáta v textovom formáte. Zároveň si do premennej uložíme inštanciu objektu Octokit s nastavenými parametrami hlavičky, ktorú budeme využívať pri ďalšej práci s Githubom. Celú odpoveď servera spolu s premennou octokit si uložíme do lokálneho stavu aplikácie v Reduxe.

```
export const githubSignin = () =>{
    const provider = new firebase.auth.GithubAuthProvider();
    return(dispatch) =>{
        provider.addScope('repo')
        firebase.auth().signInWithPopup(provider)
        .then((result) => {
            localStorage.setItem("user", JSON.stringify(result))
            var octokit = new Octokit({
                auth: result.credential.accessToken,
                userAgent: "Mathematics4",
                baseUrl: "https://api.github.com"
            })
            dispatch({
                type: "LOGIN_USER",
                data: JSON.parse(JSON.stringify(result)),
                octokit
            })
        })
        .catch((err) => {
            dispatch({
                type: "LOGIN_USER_ERROR",
                err
            })
        })
    }
}
```

6.5 Prehľadávanie repozitárov

Dôležitou časťou našej práce je možnosť prehliadať všetky repozitáre používateľa a možnosť získavať dáta z jednotlivých súborov z Githubu. Preto tu opäť využívame knižnicu Octokit, ktorá nám uľahčuje komunikáciu s Github REST API v3 a čiastočne upravuje odpoveď, ktorú získavame.

6.5.1 Získavanie dát

Používateľ si najskôr vyberie jeden zo svojich repozitárov, tieto dáta sme získali hneď po spustení aplikácie prihláseného používateľa, prípadne po jeho prihlásení. Tu využijeme uloženú inštanciu objektu Octokit a jednoduchú funkciu, ktorej argumentom je typ repozitárov, ktoré chceme získať. V tomto prípade chceme tie, ktorých je používateľ vlastníkom a odpoveď si uložíme do lokálneho stavu v Reduxe.

```
octokit.repos.listForAuthenticatedUser({type:"owner"})
```

Podobne postupujeme aj pri vetvách používateľom vybraného repozitára, len použijeme inú funkciu.

```
octokit.repos.listBranches({
        owner: repo.owner.login,
        repo: repo.name,
    })
```

Po výbere konkrétnej vetvy sa zobrazí prehliadač repozitára, v ktorom sa nachádza zoznam súborov. Pre získanie týchto súborov opäť používame funkciu knižnice Octokit. Pri tejto funkcii zadávame do argumentov aj cestu ku konkrétnemu súboru a využívame ju pre získavanie obsahu všetkých typov súborov. Pri prvom otvorení vetvy využívame túto funkciu len v základnej podobe s povinnými argumentmi a keďže sa nachádzame v koreni, tak premennej path, ktorá určuje cestu ku súborom, priradíme prázdny reťazec.

```
octokit.repos.getContents({
        owner: currentRepo.owner.login,
        repo: currentRepo.name,
        ref: branch.name,
        path: path
    })
```

6.5.2 URL pre pohyb v repozitároch

Pre pohyb v repozitároch primárne využívame vytvorený prehliadač a omrvinkovú navigáciu, pokiaľ sa chceme vrátiť do predchádzajúcich zložiek. No v súboroch, ktoré otvoríme naším prehliadačom sa môžu nachádzať linky do konkrétnych zložiek a súborov. Keďže ide o linky, po kliknutí na ne sa nám zmení URL adresa a tým sa dostaneme do stavu, ktorý router nepredpokladal. Vedeli by sme to samozrejme vyriešiť malou úpravou cesty routra aby tieto informácie ignoroval, no my chceme túto informáciu využiť a automaticky vojsť do súboru na ktorý táto cesta ukazuje.

Aplikácia musí teda reagovať na zmeny url a vedieť tieto dáta aj spracovať. Preto využívame Router s regexovou cestou. Keďže celý prehliadač repozitárov funguje na jednej stránke, musí sa tento komponent zobraziť pri akejkoľvek URL adrese. Kvôli zlepšeniu prehľadnosti a zjednodušeniu práce v našom komponente sme využili takýto

typ cesty. Otáznikom za kľučom v ceste označíme tento kľúč za nepovinný a hviezdičkou, že ku tomuto kľúču môže byť priradených viac hodnôt.

```
<Route path = "/:repo?/:branch?/:path*" component={RepoBoard}/>
```

Takouto cestou sme docielili, že náš komponent RepoBoard, ktorý zobrazuje prehliadač repozitárov, dostane v Props údaje z URL v objekte match. Tento objekt má v sebe ďalší objekt s názvom params, v ktorom sa nám zobrazujú kľúče s hodnotami. Vďaka tomu, že sme si cestu v našom Routri rozdelili takýmto spôsobom, získame v objekte params hodnoty rozdelené podľa typu.

```
params: {
    repo: "lpi19-kniha2"
    branch: "master",
    path: "tools/win"
}
```

Listing 6.1: Objekt params pre URL "/lpi19-kniha2/master/tools/win"

Teraz vieme v našom komponente priamo pracovať s tými údajmi bez nutnosti ďalších úprav reťazca. Preto pri načítaní komponentu zisťujeme čo naša URL adresa obsahuje a podľa toho nastavíme prehliadač repozitárov.

Pre načítanie konkrétneho súboru z cesty nám však nestačí jeho cesta, pretože môže ísť o dva typy súborov:

- Zložka, pri ktorej potrebujeme zobraziť zoznam všetkých súborov, ktoré obsahuje.
- Konkrétny súbor, ktorého obsah chceme zobraziť a taktiež načítať do zoznamu celú zložku v ktorej sa nachádza.

Priamo z cesty, ktorú dostaneme nevieme určiť o aký typ súboru ide, preto potrebujeme najskôr získať dáta a podľa toho určiť čo má aplikácia robiť. Keďže funkcia `octokit.repos.getContents()` priamo neobsahuje informáciu o tom, či sú naše dáta súbor alebo zložka, musíme iba skontrolovať či sú dáta uložené ako textový reťazec alebo pole.

V prípade, že ide o pole tak je to zložka a my prvky tohto poľa zobrazíme v našom zozname a vytvoríme linky.

Ak ide o textový reťazec, tak si tieto dáta uložíme a sme pripravení ich zobraziť, no potrebujeme ešte načítať aj zložku, v ktorej sa tento súbor nachádza. Na to využijeme funkciu, v ktorej si cestu rozdelíme do poľa, zbavíme sa posledného prvku a opäť pustíme funkciu na získanie dát s cestou, pri ktorej vieme, že ukazuje na zložku.

Pre prácu s repozitármi pomocou URL sme však museli vytvárať aj vlastné adresy pre komponent Link. Chceli sme použiť rekurzívny Router, no bohužiaľ ten nefungoval, pretože typicky sa tento typ využíva pri komponentoch, ktoré sa postupne zobrazujú a

každý z nich má na starosť svoju časť URL adresy. Naša aplikácia však stále pracuje s jedným komponentom, ktorý sa musel stále znova prekresliť. Preto sú jednodlivé prvky v našom zozname komponentu Link, pre ktorý vytvárame vlastné adresy podľa toho či sa jedná o súbor alebo zložku.

```
1     e.type === "dir" ?
2             <Link to={generateFolderLink(e)}>{e.name}/</Link>
3         :
4             <Link to={generateFileLink(e)}>{e.name}</Link>
```

Listing 6.2: Objekt params pre URL "/lpi19-kniha2/master/tools/win"

6.6 Úlohy

Hlavným zameraním tejto práce je umožniť zadávanie a riešenie úloh, tieto úlohy sa zobrazujú po sebou ako prázdne bunky, s ktorými vieme ďalej pracovať. Aby sme to docielili využili sme komponent Card, ktoré sú uložene v zozname ListGroup, ktoré nám ponúka Bootstrap.

6.6.1 Získavanie a odosielanie dát

Dáta, ktoré zobrazujeme v našom komponente získavame zo súborov, ktoré sú uložené na Githube. Pre získanie dát opäť využívame funkciu `octokit.repos.getContents()`. Keďže však počas vyberania súboru, prípadne pri načítaní priamo z URL adresy nevieme povedať či ide o súbor so zadaniami, skúsime si výsledok tejto funkcie konvertovať zo súboru typu JSON na objekt. Ak sa to je úspešné, musíme ešte skontrolovať či ide o objekt so zadaniami. Preto obsahuje každý súbor na začiatku kľúč, podľa ktorého to vieme určiť. Ak je to úspešné, tak tento objekt pošleme ďalšej akciií, ktorá sa postará o správne rozdelenie zadaní a odoslanie Reduceru. Aplikácia uloží informácie o tomto súbore a v prípade, že nejde o súbor s úlohami tak zobrazí obsah tohto súboru v prehliadači repozitárov.

Pre odosielanie dát využívame inú funkciu, ktorej už musíme zadať viac parametrov. Túto funkciu využívame pri upravovaní a vytváraní nových súborov.

Github svoje dáta ukladá v Base64 kódovaní, v prípade získavania dát sa o to postará priamo knižnica Octokit, keď však chceme dáta ukladať, musíme si náš objekt s dátami zmeniť na textový reťazec a ten potom zakódovať.

```
1     var content = Base64.encode(JSON.stringify(newCardArr))
```

V prípade, že sa snažíme už existujúci súbor upraviť, je dôležitým parametrov tejto funkcie sha, teda špecifický kľúč, podľa ktorého dokáže Github tento súbor rozpoznať. V prípade, že náš sha kľúč nieje rovnaký ako ten, ktorý má súbor na zadanej ceste na Githube, funkcia nám vráti chybu a súbor sa neupraví.

```
1    octokit.repos.createOrUpdateFile({
2        owner: repo.currentRepo.owner.login,
3        repo: repo.currentRepo.name,
4        path: repo.currentFile.path,
5        branch: repo.currentBranch.name,
6        message: "...",
7        sha: repo.currentFile.sha,
8        content: content,
9        headers: {
10           'If-None-Match': ''
11       }
12   })
```

Keď sme však takto súbor upravili, úspešne odoslali, získali nový sha kľúč, ktorý sme si v našom lokálnom stave uložili a obnovili stránku, nastal problém. Funkcia na načítanie dát z Githubu nám vrátila ešte predchádzajúcu verziu súboru a pri snahe ho znova upraviť nám už nesedeli sha kľúče. Funkcia na úpravu súborov má totiž v hlavičke nastavené cachovanie v prehliadači, kvôli čomu funkcia na získanie súborov zobrala starú verziu. Toho cachovanie sa priamo vypnúť nedá, preto sme použili 'If-None-Match' trik, vďaka ktorému sa dáta dočasne neukladajú a vždy sa nahrajú nové, čím sme sa vyhli problému s nesprávnymi dátami.

6.7 Integrácia riešičov

6.7.1 Tableau Editor

Počas implementácie tohto tablovéh editora, sme museli pracovať s jazykom Elm, v ktorom je táto aplikácia vytvorená. Preto sme si museli do nášej práce pripojiť dve knižnice. Najskôr knižnica Elm, ktorá našej aplikácii umožní kompilovať kód napísaný v jazyku Elm. Druhou je React-Elm-Components, ktorá slúži na jednoduché spustenie Elmovej aplikácie ako samostatného komponentu. Keďže naša aplikácia využíva Webpack, musíme jej explicitne nastaviť čo má robiť pre súbory typu Elm. Preto využívame knižnicu React-app-rewired, ktorá nám umožňuje upraviť súbor s nastaveniami Webpacku.

Keďže potrebujeme upravovať súbor nastavení, ktorý obsahuje veľmi veľa dát, vytvoríme si súbor, v ktorom zapíšeme všetky zmeny a vložíme ich do pôvodného súboru s nastaveniami.

Prvou funkciou zabezpečíme pridanie podpory súborov typu Elm a nastavíme im iný typ webpacku, ktorý majú využívať pri spúšťaní.

Súbor so štýlmi tablovača sa nachádza v jeho zložke. Keďže nechceme stále tento súbor ručne kopírovať, nastavíme webpack aby kopíroval súbor z tejto zložky do finálnej zložky s vybudovanou aplikáciou.

```
const CopyPlugin = require('copy-webpack-plugin');
const path = require('path');

module.exports = function override(config, env){
    let loaders = config.module.rules[2].oneOf
    loaders.splice(loaders.length -1, 0, {
        test: /\.elm$/,
        exclude: [/elm-stuff/, /node_modules/],
        use: {
          loader: 'elm-webpack-loader',
          options: {}
        }
      })

    let copyPlugin = new CopyPlugin([{
      from: path.resolve('src/components/solvers/tableauEditor/src
         /tableauEditor.css'),
      to: path.resolve('build/tableauEditor.css'),
      toType: 'file',
    }])
    config.plugins = [copyPlugin, ...config.plugins]
    return config
}
```

Najskôr si potrebujeme importovať tento komponent, preto sme si všetky potrebné zdrojové súbory tejto aplikácie pridali ku naším. Tento tablovač vyžaduje vstupné Props, ktoré sa v tomto prípade nazývajú flags, ktoré potom využije na vytvorenie tabla podľa zadaných dát. Druhým argumentom sú Ports, ktoré slúžia ako Props, pomocou ktorých môže komponent posielať dáta smerom von. Keďže chceme využívať tento tablovač pri úlohách, musíme byť schopní získavať tieto dáta, následne ich uložiť a pri opätovnom načítaní zadaní ich zobraziť.

Pre nastavenie portov využívame funkciu `setupPorts`, pomocou ktorej získavame dáta a uložíme ich do karty, v ktorej sa tablovač nachádza.

```
const setupPorts = (ports) => {
    ports.print.subscribe(function () {
     });
    ports.cache.subscribe(function (tableauData) {
        handleChange(tableauData)
    });
}
```

Po nastavení portov môžeme využiť komponent knižnice React-Elm-Components, ktorý do argumentu dostane cestu k nášmu tablovaču a ak máme uložené dáta k tomu tablovaču, tak ich vložíme do flags, aby sa nám nevytvorilo tablo prázdne, ale v tvare, v akom sme ho uložili.

```
<div className="tableauEditor-container">
    <Elm src={Editor.Elm.Editor} flags={content ? content : null}
    ports = {setupPorts}></Elm>
</div>
```

Tento tablovač má vlastné štýly nastavené vo svojom css, mohlo by sa nám stať, že nám tento css súbor ovplyvní aj prvky našej aplikácie. Preto celý tento komponent obalíme do vlastnej triedy a jej názov pridáme aj ku všetkým definicíciam v css súbore.

6.7.2 Resolution Editor

Tento editor je napísaný rovnako ako naša aplikácia v Reacte s využitím Reduxu. Implementácia preto nevyžadovala špeciálne knižnice ako Tableau Editor. Aplikáciu sme sa snažili implementovať čo najvšeobecnejším spôsobom, aby bolo v budúcnosti jednoduché pridať ďalšie podobné aplikácie vytvorené v Reacte.

Do nášho projektu sme si vložili všetky zdrovojé súbory aplikácie a pridali k nim jeden náš súbor, ktorý zabezpečí spoluprácu medzi aplikáciami. Tento súbor nahrádza index.js, v ktorom sa pôvodne aplikácia inicializovala. Náš komponent s názvom embed.js uložíme na rovnaké miesto ako index.js aby ostala zachovaná štruktúra. Tento komponent potom exportujeme a použijeme ako typický komponent.

```
<Container fluid>
    <ResolutionEditor initState = {content} changeState = {
        handleChange}></ResolutionEditor>
</Container>
```

Ako argumenty mu nastavíme počiatočný stav a funkciu, ktorá bude ukladať aktuálny stav tejto aplikácie v našom stave, aby sme ho mohli uložiť na GitHub a po opätovnom otvorení tieto dáta načítať.

Pre vloženie počiatočného stavu využijeme nepovinný argument funkcie na vytvorenie Redux Store, ktorý dáta nastaví ako stav v prípade, že mu nejaké dáta naša aplikácie pošle, ak nie, tak sa vytvorí prázdny stav.

```
var store = null
if(initState === ""){
    store = createStore(
        reducer,
        applyMiddleware(sender)
    );
}else{
    const preloadedState = importState(initState)
    store = createStore(
        reducer,
        preloadedState,
        applyMiddleware(sender)
```

```
        );
    }
```

Pre ukladanie stavy využívame našu funkciu, ktorú sme poslali ako argument, no potrebujeme získať prístup k najnovším dátam po vykonaní zmien. Na to na slúži vlastný middleware, teda funkcia, ktorá vykoná akciu no pred odoslaním do Store môže vykonať dodatočné zmeny. Takýto middleware má prístup ku funkcii getState(), ktorá nám vráti aktuálny stav, tento stav si pomocou funkcie exportState() riešiča upravíme do vhodnej formy a uložíme v našom lokálnom stave. Pre zníženie záťaže tieto dáta budeme ukladať iba do určitých typoch akcií.

```
    const sender = ({getState}) => {
        return next => action => {
            const returnValue = next(action)
            if(action.type === "INPUT_BLUR" || action.type === "
                ADD_STEP" || action.type === "DELETE_STEP"){
                changeState(exportState(getState()))
            }
            return returnValue
        }
    }
```

Kapitola 7

Testovanie

7.1 Cieľ testovania

Testovaním sme chceli zistiť intuitívnosť používateľského prostredia pre študentov, ktorí s aplikáciou nemajú skúsenosti. Tatktiež sme chceli odhaliť chyby a nedostatky, ktoré neboli odhalené počas vývoja.

7.2 Priebeh testovania

Aplikácia bola predstavená študentom 2. a 3. ročníka na predmete Matematika 4 - Logika pre informatikov. Pre účely testovania sme pripravili niekoľko testovacích zadaní úloh priamo v aplikácii, ktoré pre riešenie využívajú všetky typy kariet. Študenti nedostali pred testovaním žiadne inštrukcie aby sme overili jednoduchosť aplikácie. Po skončení testovania sme študentov požiadali o vyplnenie krátkeho dotazníka s otázkami.

7.3 Výsledky testovania

1. Bolo používateľské rozhranie aplikácie dostatočne intuitívne ?
2. Vyhovovala vám rýchlosť odozvy aplikácie ?
3. Využili by ste pre výučbu radšej túto aplikáciu alebo doterajší spôsob riešenia ?
4. Akú funkcionalitu by ste do aplikácie radi pridali ?

Záver

Úspešne sme vytvorili funkčnú, stabilnú a užitočnú aplikáciu, ktorá spĺňa zadané ciele. Táto aplikácia bude pravidelne využívaná pri výučbe predmetu a odovzdávaní teoretických úloh čo študentom spríjemní a uľahčí prácu na predmete.

Prácou na tejto aplikácii som mal možnosť naučiť sa prácu s novými technológiami, primárne React a Redux, ktoré som doteraz nevyužíval. Taktiež som sa naučil vytvárať jednoduché a intuitívne používateľské prostredie, navrhovať celkovú štruktúru aplikácie a získal som skúsenosti pre budúci vývoj nielen webových aplikácií.

Aplikácia je naprogramovaná v jazyku JavaScript s pomocou knižnice React, ktorá je pre takýto typ aplikácie vhodnou voľbou, pretože sme vďaka nemu ušetrili veľa času. React sme podporili viacerými knižnicami ako napríklad Redux, ktorý nám pomohol zachovať modularitu a nezávislosť komponentov, vďaka čomu bude jednoduchšie aplikáciu v budúcnosti rozšíriť.

Jednou z možností pre rozšírenia je správa celého predmetu, hodnotenie študentov, novinky ohľadom predmetu a podobne.

Druhou dôležitou možnosťou rozšírenia je zapojenie ďalších matematických riešičov. Aplikácia aktuálne umožňuje využívať 2 druhy riešičov, no je pripravená na zapojenie aj iných, ktoré však budú potrebovať viac zásahov do kódu.

Literatúra

[1] Nalaka Dissanayake and Kapila Dias. Rich web-based applications: An umbrella term with a definition and taxonomies for development techniques and technologies. *International Journal of Future Computer and Communication*, 7, 03 2018.

[2] OpenJS Foundation. *Node.js*. Joyent, Inc., `https://nodejs.org/en/`.

[3] Peter Gainsford. *The title of the work*. The organization, The address of the publisher, 3 edition, 7 1993. An optional note.

[4] Google. *Firebase*. `https://firebase.google.com/`.

[5] Mosh Hamedani. *React Virtual DOM Explained in Simple English*. `https://programmingwithmosh.com/react/react-virtual-dom-explained/`, 2018.

[6] Espen Hovlandsdal. *react-markdown*. `https://github.com/rexxars/react-markdown`.

[7] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

[8] MathWorks. *MATLAB*. `https://www.mathworks.com/products/matlab.html`.

[9] Ben Straub Scott Chacon. *Pro Git*, volume 05 of *13*. `https://git-scm.com/book/en/v2`, 2.1.225 edition, 2020.

[10] Bootstrap team. *Bootstrap*. `https://getbootstrap.com/`, 4 edition.

[11] Univerzita Komenského v Bratislave. Vnútorný predpis č. 12/2013, smernica rektora Univerzity Komenského v Bratislave o základných náležitostiach záverečných prác, rigoróznych prác a habilitačných prác, kontrole ich originality, uchovávaní a sprístupňovaní na Univerzite Komenského v Bratislave, 2013. `https://uniba.sk/fileadmin/ruk/legislativa/2013/Vp_2013_12.pdf`.