SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-92671

WEBOVÁ APLIKÁCIA NA SPRÁVU STREŠNÝCH STAVIEB

BAKALÁRSKA PRÁCA

2021 **Tomáš Vago**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-92671

WEBOVÁ APLIKÁCIA NA SPRÁVU STREŠNÝCH STAVIEB

BAKALÁRSKA PRÁCA

Študijný program:	aplikovaná informatika
Názov študijného odboru:	Informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Michal Kocúr, PhD.

Bratislava 2021 **Tomáš Vago**

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
2020/2021

ZADANIE BAKALÁRSKEJ PRÁCE

Autor práce:	Tomáš Vago
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Evidenčné číslo:	FEI-5382-92671
ID študenta:	92671
Vedúci práce:	Ing. Michal Kocúr, PhD.
Miesto vypracovania:	Ústav automobilovej mechatroniky
Názov práce:	**Webová aplikácia na správu strešných stavieb**
Jazyk, v ktorom sa práca vypracuje:	slovenský jazyk

Špecifikácia zadania:

Cieľom bakalárskej práce je navrhnúť webovú aplikáciu, ktorá dokáže na základe vstupných údajov vypočítať množstvo stavebného materiálu a ďalšieho príslušenstva na stavbu strechy z PVC fólie. Na základe tejto kalkulácie bude aplikácia generovať cenovú ponuku a spravovať proces prípadnej objednávky. Súčasťou aplikácie bude aj správa skladových zásob stavebného materiálu, ktorá bude prepojená so systémom objednávok.

Úlohy:
1. Na základe analýzy vyberte vhodné technológie na vytvorenie webovej aplikácie.
2. Navrhnite štruktúru informačného systému s aplikačným rozhraním na základe požiadaviek REST.
3. Navrhnutý systém implementuje prostredníctvom naštudovaných programovacích jazykov.
4. Overte funkčnosť vytvorenej aplikácie.
5. K vytvorenej aplikácií zhotovte užívateľskú príručku.

Literatúra:

MySQL, MySQL Documentation [online]. 2019. URL: https://dev.mysql.com/doc/
SUBRAMANIAN, Harihara and RAJ, Pethuru. Hands-on RESTful API design patterns and best practices: design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs. Birmingham, UK, PACKT Publishing Limited, 2019.

Dátum zadania: 15. 02. 2021

Dátum odovzdania: 04. 06. 2021

Tomáš Vago
študent

Dr. rer. nat. Martin Drozda
vedúci pracoviska

Dr. rer. nat. Martin Drozda
garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	aplikovaná informatika
Autor:	Tomáš Vago
Bakalárska práca:	WEBOVÁ APLIKÁCIA NA SPRÁVU STREŠNÝCH STAVIEB
Vedúci záverečnej práce:	Ing. Michal Kocúr, PhD.
Miesto a rok predloženia práce:	Bratislava 2021

Cieľom tejto bakalárskej práce je naštudovanie vhodných technológii, ktoré budú použité na vytvorenie webovej aplikácie pre strechársku firmu, ktorá za pomoci vstupných parametrov vypočíta cenovú ponuku, ktorá bude následne uložená v databáze kvôli archivácii a zároveň vygenerovaná ako excel súbor. Sekundárnou funckionalitou tejto aplikácie je správa skladu pre strechársku firmu. V tomto sklade je možné pracovať s položkami, ktoré sa v ňom nachádzajú a meniť ich atribúty. Všetky tieto akcie sú vykonávanie za použitia REST metód, ktoré sú implementované v Jave za použitia frameworku Spring a v JavaScripte za použitie frameworku React.

Kľúčové slová: webová aplikácia, informatika

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Tomáš Vago
Bachelor's thesis:	Roof Construction Management Web Application
Supervisor:	Ing. Michal Kocúr, PhD.
Place and year of submission:	Bratislava 2021

The aim of this bachelor thesis is to study the appropriate technologies that will be used to create a web application for a roofing company, which uses input parameters to calculate a price offer, which will then be stored in a database for archiving and generated as an excel file. The secondary functionality of this application is warehouse management for a roofing company. In this warehouse, it is possible to work with the items that are stored in it and change their attributes. All of these actions are performed using REST methods, which are implemented in Java using the Spring framework and in JavaScript using the React framework.

Keywords: web application, information technology

Poďakovanie

Chcel by som poďakovať pánovi Ing. Michal Kocúr, PhD. za jeho trpezlivosť, ochotu a poskytnutie odborných informácii a rád, ktoré mi vo veľkej miere pomohli k úspešnému zhotoveniu tejto práce.

Obsah

Zoznam obrázkov a tabuliek

Úvod

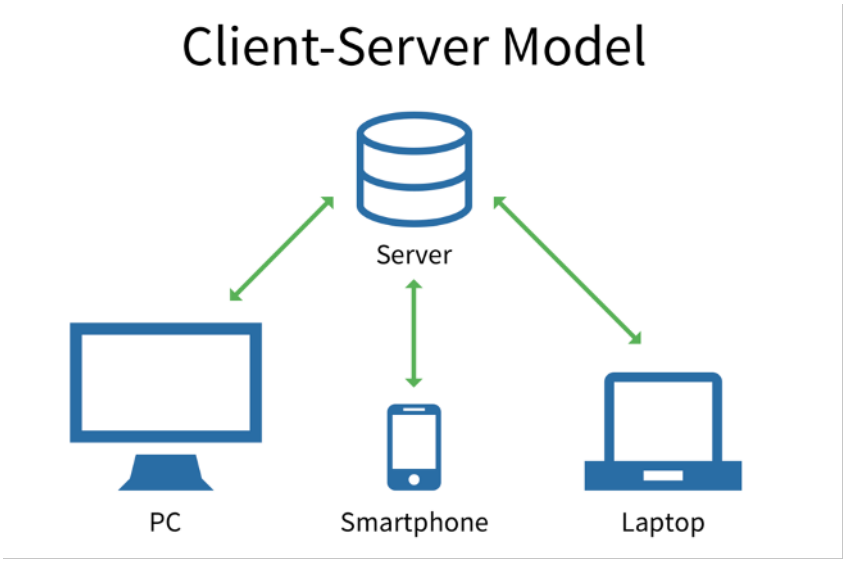
Veda a technika deň čo deň napredujú míľovými krokmi a pomáhajú nám stále čoraz viac zjednodušovať dennodennú prácu, ktorú častokrát vykonávame už takmer automatizovane, no neuvedomujeme si, že sme ľudia a nie roboty. Práve rôzne technické vylepšenia sú tu pre nás, aby sme ich využívali a nestrácali čas a energiu pri aktivitách, ktoré môžu pre nás znamenať len zopár klikov. Ľudia pracujúci v stavebníctve často musia robiť počas vypracovávaní cenových ponúk rôzne drobné výpočty, ktoré síce nie sú náročné no zaberú veľa času a preto je jednou z hlavných úloh tejto práce vypracovať návrh a implementovať webovú aplikáciu, ktorá dokáže po zadaní rozmerov strechy vypočítať množstvo potrebného materiálu a vytvoriť cenovú ponuku, ktorá prejde celým svojim workflowom, ktorý bude ďalej popísaný v našej práci. Celkový systém by mal ďalej užívateľovi pomáhať efektívne spravovať a využívať skladové zásoby, ktorými firma disponuje. Aplikácia bude pri každej cenovej ponuke vyhodnocovať, či jednotlivé položky potrebné na zhotovenie strechy sú dostupné v sklade, alebo je potrebné ich dokúpiť. Ďalšou pomocnou funkcionalitou bude archivácia jednotlivých cenových ponúk, ktoré budú v systéme evidované so svojim špecifickým statusom, ktorý bude cenové ponuky triediť do skupín. Samotnému vypracovaniu tejto práce predchádza dlhoročná praktická skúsenosť v oblasti stavebníctva a teda aplikácia poskytuje službu, ktorá bude po zhotovení aktívne využívaná externou firmou, s ktorou sme počas celej implementácie neustále komunikovali.

1 Analýza základných pojmov

V tejto kapitole sme sa rozhodli venovať analýze potrebných pojmov, ktoré využívame v našej práci na riešenie jednotlivých úloh, či problémov. Tvorba webovej aplikácie sa zakladá na správnom použití jednotlivých technológií, z ktorých po správnom použití vznikne jeden veľký ucelený systém, ktorý bude stabilný a nebudú sa v ňom nachádzať žiadne chybné časti či nedostatky. V našej práci sme sa rozhodli využiť tie najoverenejšie a najčastiejšie používané technológie, ako napríklad: REST api, databáza, Git.

1.1 Klient-server model

Klient-server model popisuje ako server poskytuje dáta a služby pre jedného alebo viacerých klientov. Medzi najčastejšie druhy serverov, ktoré používame patria webové, mailové alebo dokumentové. Každý z týchto serverov zaručuje, že zdroje z nich sa dostanú do používateľských zariadení. Väčšina týchto serverov sú v relácii ku klientom 1:N, čo znamená, že jeden server môže spolupracovať s viacerými klientmi súčasne.



Obr. 1: Schéma ku klient-server modelu

1.2 REST API

REST API (Representational state transfer - prevodník dát) je aplikačné programátorské rozhranie, ktoré vyhovuje obmedzeniam štýlu restovej architektúry a umožňuje komunikáciu s REST webovými službami. V laickej reči možno vysvetliť tento pojem ako dohodu medzi niekým, kto bude spracovávať informácie (server) od niekoho, kto si ich vyžiada (klient-request) a vďaka tejto dohode vznikne vyprodukovaný spracovaný obsah informácii

(response). Jednoducho povedané, ak chcete komunikovať so serverom alebo akýmkoľvek systémom kvôli získaniu dát, alebo vykonaniu nejakej funkcie, API pomáha s touto komunikáciou, aby systém mohol pochopiť a vykonať vašu požiadavku.

1.3 Databáza

Databáza je organizovaná zbierka dát, prípadne informácií, ktoré sú zväčša uložené elektronicky v počítačovom systéme. [1] Vďaka databáze je v našej práci možné ukladať rôzne cenové ponuky, dáta o jednotlivých strechách, prípadne konkrétnych zákazníkoch, či na druhej strane správa skladov a ich naskladnených zásob. Používali sme databázu mySQL, pretože sa jedná o relačnú databázu, čo pre nás znamená oveľa prehľadnejší systém, keďže pre jednotlivé objekty, prípadne mapované entity ukladáme separátne do osobitných tabuliek.

1.3.1 Relačná a nerelačná databáza

Poznáme dva typy databáz a to relačnú a nerelačnú. Hlavným rozdielom medzi nimi je, že relačná databáza obsahuje tabuľky pre jednotlivé databázové entity, ktoré majú medzi sebou vzťahy 1:n , 1:1 alebo n:n. Nerelačná databáza sa neukladá do tabuliek, ale jej dáta sú ukladané ako kľúč a k nemu príslušná hodnota.

- 1:1 relácia - ide o reláciu, pri ktorej vzťah medzi dvomi objektami nevyžaduje mapovaciu tabuľku. Pri tomto mapingu sa využíva pridanie cudzieho kľúča tabuľkám, ktoré patria dvom asociovaným objektom. Príklad pre takýto vzťah je v našom prípade cenová ponuka a strecha. Jedna cenová ponuka môže byť namapovaná len na jednu strechu a pre náš systém môže v databáze byť strecha pre ktorú existuje len jedna cenová ponuka.

- 1:n relácia - ide o reláciu, pri ktorej vzťah medzi dvoma objektami nevyžaduje mapovaciu tabuľku. Podobne, ako pri 1:1 relácii je vyriešený mapovanie medzi dvoma tabuľkami pomocou cudzieho kľúča, avšak v tomto prípade nie je cudzí kľúč unikátna hodnota (môže sa vyskytovať pre daný stĺpec vo viacerých riadkoch). Príklad v našom systéme môže byť napríklad strecha a komín. Na jednu strechu v databáze môže byť namapovaných viacero komínov, avšak jeden komín môže byť namapovaný len na jednu strechu.

- n:n relácia - ide o reláciu, kedy je nevyhnutné použitie mapovacej tabuľky. Táto tabuľka je tvorená použitím idčkami oboch objektov. V našom systéme môžeme ako príklad uviesť entitu užívateľ a entitu autorita. Jeden užívateľ môže patriť do viacerých autorít, ale rovnako aj autorita môže vlastniť viacerých rôznych užívateľov.

Medzi používané relačné databázy patria napríklad MySQL alebo PostgreSQL. Relačné databázy využívame najčastejšie pri webových aplikáciach, ktoré potrebujú viacero tabuliek s definovanými vzťahmi medzi nimi.

Nerelačné databázy sa využívajú najmä v prípade, že sa do databázy ukladá veľmi veľké množstvo dát, v ktorom užívateľ prípadne potrebuje rýchle vyhľadávanie. Môžu byť tvorené pomocou dokumentov (JSON), grafov alebo kľúčov a hodnôt.

1.4 Frontend a backend

Frontend počítačového programu, webovej aplikácie alebo stránky je všetko, s čím môže používateľ prísť do kontaktu. Z pozície užívateľa je to v istom slova zmysle používateľské rozhranie. Je veľmi dôležité, aby bol frontend každej webovej aplikácie čo najviac priehľadný, pochopiteľný a najmä intuitívny. Z technického hľadiska frontendová časť obsahuje stránky, grafické nastavenia a funkcie, textový obsah, rôzne tlačidlá, vstupné textové polia a podobne. Backend je v počítačovom svete akákoľvek časť aplikácie, či programu, ktorú užívateľ, na rozdiel od frontendovej časti, nevidí. V programátorskom ponímaní ide o vrstvu pracujúcu s dátami, pričom o frontende môžeme hovoriť ako o prezentačnej vrstve. Všetko, čo sa udeje pred tým, ako na výstupe niečo používateľ môže vidieť, je práca backendu. Backend spravuje prichádzajúce requesty, rôzne scipty, prácu s databázou, šifrovanie alebo dešifrovanie dát, upload alebo download dokumentov a podobne. Všetky tieto spomenuté príklady prebiehajú na strane serveru. Backend a frontend spolu spolupracujú na vytvorení komplexného produktu pre používateľa.

2 Frameworky

V programátorskej terminológií chápeme pojem framework ako predpripravenú istú skupinu riešení, ktoré pomáhajú používateľovi riešiť základné, bežné programátorské problémy. [2] Pre každého profesionálneho programátora su takmer nevyhnutným spoločníkom pri práci, pretože okrem toho, že pomáhajú k jednoduchosti, veľkou mierou prispievajú ku rýchlosti implementácie daného produktu. V našej bakalárskej práci sa stretneme s frameworkami spring a hibernate na strane backendu a react na strane frontendu.

2.1 Frameworky využívané na tvorbu backendu

Na programátorskom trhu je pomerne veľké množstvo používaných jazykov a frameworkov, pomocou ktorých sa dá vytvoriť finálová aplikácia. Každé z nich má svoje výhody, aj nevýhody. My sme sa rozhodli pred zvolením frameworku tieto výhody a nevýhody pomerne viac rozobrať a nakoniec sme sa rozhodli zvoliť na backendovej časti Spring, ktorý sa programuje v Jave. V tejto podkapitole si všetky rozoberané bližšie popíšeme.

- Laravel. Jedná sa o framework, ktorý sa programuje v jazyku PHP. Jeho syntax nám v porovnaní so Springom prišla pomerne jednoduchá, spustenie samotného projektu bolo omnoho jednoduchšie, nakoľko spring vyžaduje určitú konfiguráciu. Avšak napriek tomu nás laravel odradil jeho obmedzeným množstvom funkcii, ktoré ponúka. Ďalšou pomerne veľkou nevýhodou PHP je, že požiadavky od klienta sa vykonávajú postupne. Znamená to, že nerozlišuje náročnosť jednotlivých požiadaviek. Užívateľ odošle požiadavku na server, PHP alokuje potrebnú pamäť a neuvoľní ju, kým sa proces nedokončí.

- Node.js. Ide o framework využívaný opäť na strane servera. Jedná sa o framework, ktorý je navrhnutý na používanie len za pomoci jedného vlákna, čo vyžaduje využívanie asynchrónneho programovania. [3] V prípade, že by sme v budúcnosti chceli náš systém rozšíriť o väčšie množstvo výpočtov, ktoré by na pozadí bežali za pomoci viacerých jadier počítača a záviseli by tie jednoduchšie od tých komplikovanejších, nebol by Node pre nás vhodný. Ďaľšou nevýhodou, ktorá je o Node pomerne známa v kruhoch programátorskej verejnosti je nestabilita API pri aktualizovaní verzii. V mnoho prípadoch je potrebné prerábať celé časti kódu. Tento framework je využívaný najmä pre softvér, ktorý musí zobrazovať dáta v reálnom čase, napríklad rôzne chatovacie aplikácie.

2.2 Frameworky využívané na Frontende

Rovnako ako pri výbere vhodného frameworku pre frontendovú časť, sme mali k dispozícii viacero dostupných, ktoré sa aktuálne často používajú. My sme si do užšieho výberu vybrali tri. React.js, Angular.js a Vue.js . Všetky tieto frameworky sú aktuálne používané developermi po celom svete, React napríkad využíva spoločnosť Facebook. Angular sa rozhodli využívať obrovské spoločnosti na vývoj PayPalu, či Gmailu a rovnako nezaostáva ani Vue, ktoré sa využíva v Nintende.

- Angular. Skladá sa zo siedmich vydaných verzí od Angular 2 po Angular 8. Angular, ktorý je mimochodom vyvinutý firmou Google

2.3 Spring

Jedná sa o jeden z najpopulárnejších frameworkov v oblasti vývoja aplikácii pre programovací jazyk Java. [4] Používajú ho milióny vývojárov po celom svete najmä kvôli jeho výkonu a celkovej stabilite. V neposlednom rade sú jeho ďalšími výhodami možnosť ľahkého testovania kódu pomocou napísaných unit testov a prenosnosť kódu. Tento framework je celý písaný v Jave a jedná sa o open-source. Jeho najdôležitejšími funkciami a zároveň výhodami sú:

- Veľkosť - Spring so všetkými jeho knižnicami má pomerne malú veľkosť, ak sa pozrieme na to, ako veľa služieb poskytuje.
- Pokrýva princípy objektovo orientovaného programovania.
- Je možná integrácia s rôznymi inými frameworkmi.

Nevýhod podľa nás a podľa názorov na voľne dostupných programátorských [5] portáloch, či fórach je pomerne málo. Medzi najčastejšie spomínané patria:

- Prvou, azda najzákladnejšou nevýhodou je jeho zložitosť. Ak programátor používajúci spring nie je obkolesený skúsenými programátormi, ktorí sa venujú springu niekoľko rokov, nie je schopný vyžiť efektívne jeho funkcionalitu. Spring poskytuje toho naozaj veľa, no je potrebné vedieť, ako všetko používať.
- Veľké množstvo funkcii, ktoré sú vhodné na riešenie jedného problému. Ak programátor nevie, ktorá by bola najvhodnejšia a použije nesprávnu, je možné, že zložitosť nejakej operácie bude horšia, akoby použil správne riešenie. Jednoduchý príklad: Ak užívateľ chce z backendu vrátiť len nejaký atribút daného objektu a rozhodne sa vrátiť celý objekt, na odpoveď zo servera môže client čakať dlhšie.

2.4 React

Framework React.js je aktuálne jeden z najpoužívanejších frameworkov, ktorý slúži na vývoj frontendu webových aplikácii. React je open-source, čo v preklade znamená, že kód tejto knižnice je voľne dostupný a po schválení ho môže ktokoľvek upravovať. Táto knižnica je celá napísaná v jazyku JavaScrip. Používame ju na vytvorenie kompletného užívateľského rozhrania našej aplikácie. React bol prvý krát vytvorený Jordanom Walkerom, ktorý bol programátor pracujúci pre spoločnosť Facebook. Aktuálne je využívaný pre webové aplikácie ako napríklad facebook, či instagram. Výhody, ktoré určite treba tomuto pomerne mladému frameworku vyzdvihnúť sú:

- Prirodzenosť. React používa špeciálnu syntax, ktorú nazývame JSX. Táto programátorovi umožňuje používať v jednom kóde súčasne HTML aj JavaScript kód. Nie je to však nevyhnutné. Programátor môže stále písať v obyčajnom JavaScripte no jednoznačne je jednoduchšie používať JSX.
- Strmá krivka učenia. Veľká prednosť tohoto frameworku je, že stačí pomerne malý čas na pochopenie jeho princípov.
- Natívny prístup. React je možné použíť na vytvorenie mobilnej či webovej aplikácie. Dokonca je možné písať kód, ktorý bude použíteľný v iOS, Android alebo webovej aplikácii súčasne.
- Testovateľnosť. Aplikácie písané v ReactJS je veľmi jednoduché testovať či debugovať, nakoľko je možné zobraziť všetky akcie, funkcie alebo poremenné.

Nevýhod reactu existuje naozaj veľmi málo, avšak všade sa nájde niečo, čo je možné zlepšiť a rovnako to platí aj o tomto frameworku.

- Veľmi slabá dokumentácia. Je to však bežný problém softwéru, ktorý podlieha neustálym aktualizáciam technológii.
- Pokrýva výhradne časť užívateľského rozhrania, nič iné.

2.5 Hibernate

Hibernate je open source framework, ktorý poskytuje mapovanie objektových relácii do databázy pre webové aplikácie. Hibernate ORM nástroje mapujú Javovské triedy do tabuliek databázy a všetky typy jednotlivých atribútov v Jave na dátove typy SQL. Ďalej umožňuje prístup k týmto dátam vďaka vytvoreným selectom (dopytom z databázy).

Rovnako je možné dáta to databázy ukladať, vďaka použítiu repozitárov. Vo všeobecnosti ide o veľmi veľkého pomocníka, ktorý zaručuje absolútne jednoduchú komunikáciu databázy a našej webovej aplikácie. Výhod používania hibernatu máme opäť veľké množstvo, spomenieme si zopár významných:

- Takzvané lenivé načítavanie dát. Ide o spôsob, pri ktorom sa načítavajú iba potrebné dáta, ku ktorým v našej aplikácii pristupujeme. V prípade, že ich v kóde nijak nevyužívame, v databáze sa zbytočne nemusia hľadať. Pomáha to najmä k oveľa rýchlejšej dostupnosti dát.

- Používanie takzvanej hibernate query language. Tento spôsob dopytu dát je omnoho jednoduchší, ako písanie bežných sql dopytov a vo väčšine prípadov oveľa efektívnejší.

- Samozrejme je dôležité vyzdvihnúť jednoduchosť tohoto frameworku. Jeho naštudovanie a osvojenie jeho základných princípov je pomerne jednoduché a rýchle. Samozrejme, podrobné naštudovanie zaberie nejaký čas.

Taktiež aj pri tomto frameworku ťažko hľadať akési nedostatky. Tie, ktoré by sme možno ocenili, ak by v budúcnosti boli doimplementované sú:

- Nepodporuje vkladanie viacerých objektov jedným príkazom. Napriek tomu, že sa jedná o vkladanie do jednej tabuľky, je každú takúto operáciu nevyhnutné vykonávať separátne.

- Nie je najvhodnejší pre malé projekty. Nakoľko je potrebné vcelku veľkú počiatočnú konfiguráciu, kvôli ušetreniu času môže spôsobiť viac škody ako osohu.

3 Funkčné a nefunkčné požiadavky

Naša webová aplikácia sa skladá z dvoch základných častí. Prvú časť nazývame výpočtová a druhú skladová.Vo výpočtovej časti musí používateľ zadať nacenenie strechy. Následne zadá vstupné parametre: šírku strechy, dĺžku strechy, vnútornú výšku atiky a vonkajšiu šírku atiky. Ďalej musí pridať každý komín, ktorý sa nachádza na streche spolu s jeho rozmermi. Na základe týchto údajov systém vypočíta počet potrebných jednotlivých položiek: balíky krytinovej fólie, rohové lišty, šróby, okapové plechy. Podľa jednotlivých vyrátaných položiek sa vytvorí cenová ponuka, ktorá je vyexportovaná do xls predpripraveného súboru.

3.1 Funkčné požiadavky

- Možnosť prihlásenia - po spustení aplikácie sa užívateľovi zobrazí obrazovka, kde sa vyskytujú dve kolonky. Do jednej kolonky užívateľ zadá prihlasovacie meno, do druhej prihlasovacie heslo a po kliknutí na tlačidlo s nápisom "prihlásiť"bude užívateľ autentifikovaný, bude mu pridelený token, s ktorým sa dokáže autorizovať a bude presmerovaný na domovskú obrazovku aplikácie.

- Možnosť zadania vstupných parametrov strechy - užívateľ musí mať možnosť v procese vytvárania cenových ponúk zadať do vstupných koloniek číselné hodnoty, ktoré zodpovedajú nameraným parametrom naceňovanej strechy.

- Vypočítanie cenovej ponuky na základne vstupných parametrov - aplikácia musí byť schopná vypočítať cenovú ponuku, ktorá je následne uložená do databázy pre ďalšie použitie.

- Evidovanie statusu cenovej ponuky - každá cenová ponuka musí evidovať svoj status, pomocou ktorého sa dajú jednotlivé ponuky filtrovať a rozlišovať.

- Vyexportovanie cenovej ponuky do xls = implementovaná aplikácia musí vyexportovať cenovú ponuku do predpripraveného excelovského súboru, ktorý je možné odoslať zákazníkovi na posúdenie.

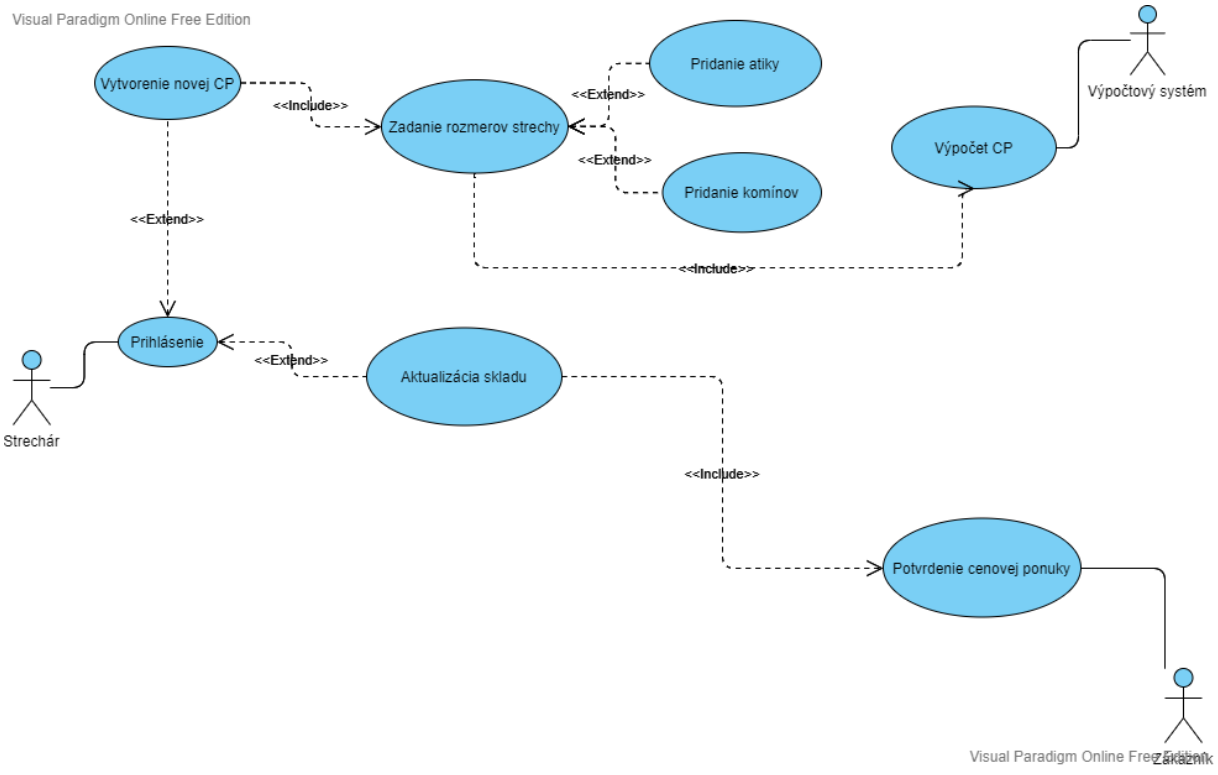
- Možnosť spravovať dostupné položky v sklade - aplikácia musí ponúkať možnosť pridávať, odoberať položky do skladu, či upravovať ich množstvo v sklade.

- Automatické odrátanie položiek pri zmene statusu cenovej ponuky na schválená. - všetky položky, ktoré sa nachádzajú na schválenej cenovej ponuke sa odpočítajú zo skladu.

3.2 Nefunkčné požiadavky

- Backend systému je napísaný v jazyku Java
- Frontend systému je napísaný v jazyku JavaScript
- Systém backendu bude využívať framework Spring
- Systém frontendu bude využívať framework React
- Cenová ponuka bude systémom vypočítaná za menej ako 3 sekundy
- Systém bude využívať databázu MySQL
- Systém vyžaduje lokálny úložný priestor pre exportovanie cenových ponúk.

3.3 Use case diagram



Obr. 2: Use case diagram

V našom diagrame použitia vystupujú tri role. Strechár, zákazník a výpočtový systém. Všetko v našom systéme je podmienené prihlásením.

4 Implementačná časť

V kapitole frameworky sme sa venovali rôznym technológiam, ktoré sme v našej práci použili na vytvorenie serverovej časti, ktorú nazývame aj backend aplikácie a pužívateľského rozhrania, ktoré nazývame frontend. Nakoľko sme počas celej implementácie chceli frontendovej časti chceli pracovať s reálnymi serverovými odpoveďami, začali sme implementáciou backendu. Veľký dôraz sme počas našej práce kládli na správne dodržanie REST princípov. Spomenuli by sme tie najhlavnejšie:

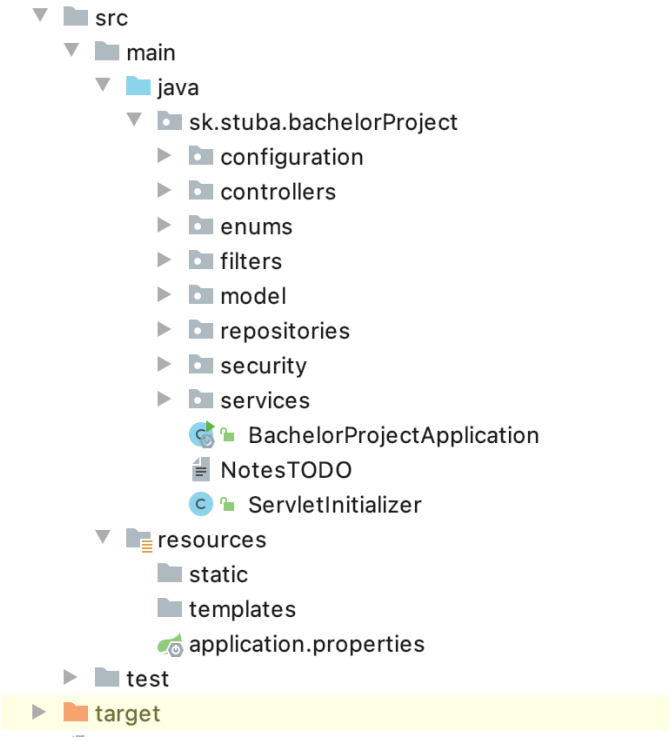
- HTTP metódy typu POST sa používajú na perzistovanie objektov do databázy. Objekt, ktorý chceme uložiť sa posiela v tele požiadavky, ktorú frontend posiela na server.
- HTTP metódy typu PUT sa používajú na upravenie objektu, ktorý sa už nachádza v databáze. V URI sa posiela id upravovaného objektu a v tele požiadavky sa posiela celý objekt s upravenými dátami.
- HTTP metódy typu GET sa používajú na získavanie objektov z databázy. Id získavného objektu sa posiela v URI.
- HTTP typu DELETE sa používajú na vymazávanie objektov z databázy. Rovnako ako pri GET metódach, sa posiela id objektu, ktorý chceme vymazat v URI.

4.1 Backendová časť

V tejto časti si popíšeme štruktúru backendovej časti aplikácie (tie najhlavnejšie časti) a jednotlivé kroky počas implementácie.

4.1.1 Štruktúra projektu

V tejto kapitole si popíšeme bližšie celkovú štruktúru nášho projektu, jednotlivé balíky, v ktorých sa nachádzajú naše triedy na strane backendu.



Obr. 3: Štruktúra backendovej časti projektu

- **Balík configuration -** V tomto balíku sa nachádzajú triedy, ktoré sú zodpovedné za správnu konfiguráciu a prvotnú inicializáciu jednotlivých objektov aplikácie napríklad ako základného používateľa, pomocou ktorého sa vieme prihlásiť, predvolený sklad, či predvolené položky, ktoré sú potrebné na výpočet strechy. Ďalej sa tu nachádza konfigurácia pre autorizáciu požiadaviek, ktoré bude server príjmať z frontendu.

- **Balík controllers -** Balík controllers obsahuje jednotlivé endpointy, ktoré sú volané z užívateľského rozhrania-frontendu. Nenachádza sa tu žiadna logika programu, len jednotlivé HTTP metódy. Každý request, okrem prihlasovacieho, má v url ceste predponu "/api", ktorá slúži na atorizáciu requestov. Request, ktorý takúto predponu v sebe má, musí byt autorizovaný prístupovým tokenom.

- **Balík model -** Tento balík je jeden z najdôležitejších. Každá jedna trieda v ňom, zodpovedá entite, ktorá sa nachádza v databáze a je pre ňu vytvorená vlastná tabuľka. Každá takáto trieda obsahuje konštruktory, pre používané objekty v aplikácii a rovnako getre a setre.

- **Balík repositories -**Balík repositories je opäť veľmi dôležitou časťou projektu, pretože vďaka triedam, ktoré obsahuje dokážeme pristupovať k databáze pomocou

hibernatu. Každá trieda je javovský interface, ktorý rozširuje hibernate knižnicu JPARepositories.

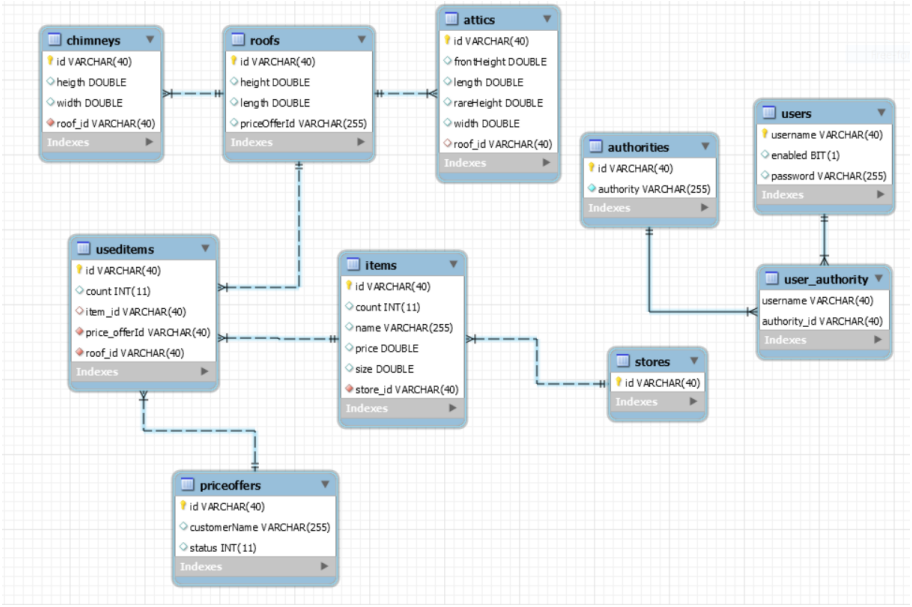
- **Balík services -** V tomto balíku, ako už z názvu ľahko rozpoznať, sa nachádzajú všetky metódy, ktoré obsluhujú na backendovej časti jednotlivé akcie, ktoré môže používateľ vykonávať. Takmer celá časť logiky, ktorá sa na pozadí odohráva je napísaná práve tu.

4.1.2 Inicializácia projektu

Ako prvú vec, ktorou sme začali bola inicializácia spring boot aplikácie. Táto aplikácia musí obsahovať základné balíky, ktoré sme popísali v predošlej podkapitole, hlavnú triedu, z ktorej sa aplikácia spúšťa a konfiguračný textový súbor, ktorý nesie koncovku .properties, v ktorom sa konfiguruje port, na ktorom chceme, aby sa aplikácia spustila, hibernate nastavenia a databázu, na ktorú sa chceme napojiť.

4.1.3 Tvorba databázy

Na základe entity diagramu sme v kóde naimplementovali časť, ktorá je zodpovedná za vytvorenie databázy. Každá entita predstavuje objekt aplikácie a podľa jednotlivých vzťahov medzi entitami sme použitím javových anotácii na určenie relácií a hibernatu vytvorili mapovacie tabuľky.



Obr. 4: EER Diagram

V tejto podkapitole si rozoberieme jednotlivé tabuľky. Budeme V našom systéme skoro každý vytvorený objekt, predstavuje entitu a zároveň každá entita má svoju vlastnú tabuľku v databáze, preto použijeme tento EER diagram na popis jednotlivých entít.

- **User** - entita user predstavuje používateľa, ktorý sa prihlasuje do systému. Jeho používateľské meno (username) je unikátne a preto je použité zároveň ako primárny kľúč tejto entity. Ďalej vlastní atribút heslo, ktoré je v databáze uložené pod šifrou, ktorú popíšeme ďalej v implementačnej časti backendu a atribút autorita, ktorý slúži počas prihlásenia k autentifikácii a zároveň ako predpríprava na možnosť pridávať užívateľovi sadu práv na jednotlivé akcie, ktoré môže vykonávať. Nakoľko medzi autoritami a používateľmi je vzťah n:n, je medzi nimi potrebná mapovacia tabuľka.

- **Roof** - entita roof predstavuje v našom systéme strechu, pre ktorú bude vytvorená cenová ponuka. Entita strecha vlastní atribút ID, ktorý je generovaný pre každú strechu a slúži ako primárny kľúč. Ďalej vlastní svoje rozmery ako dĺžka a šírka, ktoré sú definované typu double, pretože tieto rozmery môžu byť aj desatinné čísla. Ďalším atribútom je id cenovej ponuky, ktoré slúži ako cudzí kľúč odkazujúci na cenovú ponuku, ktorá je vytvorená pre danú strechu. Nasledujú atiky a komíny, ktoré sú zadefinované ako množiny objektov a ako posledným atribútom je množina položiek, ktoré musia byť nacenené. Tieto položky sú naplnené a vyrátané na základe rozmerov strechy, počtu komínov a atík.

- **Store** - entita store predstavuje sklad, v ktorom sa nachádzajú jednotlivé položky využívané počas stavby strechy. Táto entita vlastní len dva atribúty. Id, ktoré je generované rovnako ako pri všetkých ostatných objektoch a atribút položky, ktoré sú definované typu množina objektov, pretože na sklade ich môže byť 1-n. Počas tvorby cenoven ponuky sa systém pozerá do tohoto skladu a odpočítava jednotlivé položky zo systému po potvrdení cenovej ponuky zákazníkom.

- **Item** - item je entita, ktorá je využívaná v našom systéme na kalkuláciu cenových ponúk na základe ceny jednotlivých položiek v sklade. Medzi jej atribúty patrí meno položky, ktoré sa vpisuje do cenovej ponuky, ďalej veľkosť položky, jej cena za kus, sklad, v ktorom sa nachádza a počet, koľko sa danej položky nachádza na sklade.

- **UsedItem** - usedItem je entita, ktorá slúži ako doplňujúci objekt, ktorý definuje množstvo položiek použitých pre danú strechu. Tento objekt sme vytvorili ako prípravu, že systém bude môcť obsluhovať viaceré sklady naraz a v takom prípade bude nevyhnutné rozlišovanie použitých položiek a položiek, ktoré sú na sklade. Táto entita je generovaná počas kalkulácie potrebných položiek pre danú strechu.

- **Attic a Chimney** - Entita predstavujúca dopňujúce špecifické položky, ktoré môžu

byť pridané na strechu. Jej atribútmy sú rozmery jednotlivých položiek a ich rodičovská strecha.

- **PriceOffer** - entita priceOffer predstavuje cenovú ponuku.Táto entita sa vytvára v systéme až po samotnom spočítaní všetkých položiek na streche. Je ďalej využívaná ako záloha cenovej ponuky, pretože ostáva v databáze a v prípade potreby je možné z nej kedykoľvek vygenerovať excel súbor duplicitne v novej verzii, nie len prvý krát počas naceňovania. Vlastní atribúty id, ďalej meno zákazníka, ktoré je definované typu string, použité položky, ktoré sú definované ako množina objektov, pretože sa v cenovej ponuke môže nachádzať 1-n položiek. Posledným atribútom je status cenovej ponuky. Tento status sme na backendovej časti implementovali ako ENUM. Všetky statusy budú popísané v časti Služby pre obsluhu stavov cenovej ponuky.

4.1.4 Autorizácia požiadaviek na server

Na zvýšenie bezpečnosti voči vonkajším útokom na backendovú časť našej aplikácie, ktoré by sa mohli snažiť primitívne posielať požiadavky na server bez prihlásenia a tým môcť v databáze zmeniť, mazať, či pridávať údaje, je nutná autorizácia requestov. Všetky requesty sú autorizované vďaka prípone /api, ktorá určuje serverovej časti, že sa jedná o požiadavku, ktorá musí byť autorizovaná. Autorizovanie je možné vďaka tokenu, ktorý sa generuje po prihlásení autentifikáciou, ktorá bude popísaná v časti o službách prihlásenia a odhlásenia. V našej aplikácii sa nachádza len jeden request, ktorý nevyžaduje autorizáciu a tým je logicky request na prihlásenie.

4.1.5 Implementácia služieb a ovládačov

Prvé naimplementované služby boli tie, ktoré majú za úlohu vytvorenie základných objektov našej aplikácie a ich následné uloženie do databázy.

```
@Service
public class ItemService {
    @Autowired
    ItemRepository itemRepository;

    public Item createItem(Item item) {
        return itemRepository.save(item);
    }
```

Obr. 5: Príklad kódu pre uloženie novej položky do DB

Na obrázku 5 môžeme vidieť ukážku kódu, kde sa za pomoci anotácie @Autowired inicializuje repozitár, ktorý zapezpečuje prístup do tabuľky, ktorá patrí danému objektu. Pre

každý objekt, s ktorým pracujeme a ktorý má byť uložený v databáze sme napísali takúto metódu, pomocou ktorej môžeme uložiť objekt, ktorý je posielaný z používateľského rozhrania na server, do databázy.

- **Služby pre obsluhu prihlásenia a odhlásenia-** Na plnohodnotné využívanie nášho systému je potrebné prihlásiť sa vytvoreným účtom, ktorý je vytvorený pri inicializácii aplikácie. Systém autentifikuje prihlasovaného užívateľa pomocou knižnice, ktorú nám ponúka náš framework. Jedná sa o spring.security. Keďže užívateľa, ktorým sa chceme autentifikovať máme uloženého v databáze, ide o typickú oauth2 autentifikáciu. Server autentifikuje na základe posielaného mena a hesla. Heslo sa na začiatku dešifruje. Aby nebolo čitateľné a aby bola zvýšená bezpečnosť, v našej práci využívame BCEncryptor na šifrovanie hesla. Ďalej sa skontroluje, či má používateľ v databáze uloženú skupinu userov, ktorí sa môžu prihlásiť. V prípade, že je všetko v poriadku, užívateľ sa autentifikuje. Systém mu vytvorí prístupový token, ktorý máme zadefinovaný na jednu hodinu a obnovovací token, ktorý máme zadefinovaný na dve hodiny. V aktuálnej verzii systému sa využíva len prístupový token, po vypršaní ktorého bude užívateľ automaticky odhlásený a bude sa musieť opäť autentifikovať. Prístupový a obnovovací token sú počas celého systému na strane backendu uložené a nie je možné ich z užívateľského prostredia nijak vyresetovať. Na odhlásenie je potrebné zavolať DELETE HTTP metódu, ktorá vymaže vygenerované tokeny a je ich potrebné nanovo vytvoriť autentifikáciou.

```
@SuppressWarnings("unchecked")
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {

    if (Objects.equals(request.getServletPath(), b: "/oauth/token")
            && Objects.equals(request.getContentType(), b: "application/json")) {
        byte[] json = ByteStreams.toByteArray(request.getInputStream());

        Map<String, String> jsonMap = new ObjectMapper().readValue(json, Map.class);
        Map<String, String[]> parameters = jsonMap.entrySet().stream()
                .collect(Collectors.toMap(Map.Entry::getKey, e -> new String[] { e.getValue() }));

        HttpServletRequest requestWrapper = new RequestWrapper(request, parameters);
        filterChain.doFilter(requestWrapper, response);

    } else {
        filterChain.doFilter(request, response);
    }

}
```

Obr. 6: Ukážka časti kódu, ktorá spracúva POST request na prihlásenie

Aby bolo možné z frontendovej časti posielať prihlasovacie v údaje v tele požiadavky a nie v URI ceste, potrebovali sme do našej aplikácie pridať filter, ktorý spracuje json, ktorý príde v requeste a ďalej ho pošle už so správnymi dátami na ďalšie backendové spracovanie do springovej knižnice, ktorá vygeneruje už spomínané tokeny.

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {

    @Autowired
    JsonToUrlEncodedAuthenticationFilter jsonFilter;

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.addFilterAfter(jsonFilter, BasicAuthenticationFilter.class).authorizeRequests().antMatchers( ...antPatterns: "/api/**")
                .authenticated();
    }


}
```

Obr. 7: Ukážka časti kódu, v ktorej je nakonfigurovaná autorizácia

- **Služby pre obsluhu novej cenovej ponuky-** Jednou z najhlavnejších častí nášho projektu bola implementácia potrebných služieb a ovládačov, ktoré na serverovej časti aplikácie dokázali vypočítať samotnú cenovú ponuku (ďalej CP). V prvej časti výpočtu je nutná inicializácia prázdnej strechy, ktorú podrobnejšie popíšeme v implementačnej časti zameranej na frontend. Z hľadiska backendu ide o uloženie objektu strecha do databázy. Tento objekt má dva dôležité atribúty dĺžku a šírku strechy, na základe ktorých sa vypočítavajú položky potrebné pre výpočet CP. Následne takejto prázdnej inicializovanej streche dokážeme pridávať jej jednotlivé namerané atribúty, ktoré každá plochá strecha má rozličné a je potrebné ich všetky separátne pridať danej streche, pre ktorú počítame CP. Jedná sa najmä o atiky a komíny. Keďže počas tvorenia architektúry pre náš systém sme mysleli na tento fakt, objekt strecha vlastní pole objektov atiky a komíny, čo pre nás v praxi znamená, že ich vieme streche pridať dynamicky a ľubovoľný počet. Po naplnení strechy všetkými objektami, ktoré sa na nej nachádzajú sa pre vytvorenie CP, ktorá je uložená v databáze, musí zavolať služba, ktorá vypočíta nasledovné všetky preddefinované položky, ktoré musí strechár ručne počítať pre každú strechu zvlášť:

  – Množstvo potrebnej fólie. Tá sa vypočítava z celkovej plochy všetkých položiek, ktoré sa vyskytujú na streche a samozrejme z celej čistej plochy danej strechy. Celá táto plocha sa počíta v metroch štvorcových a jeden balík strešnej fólie, ktorá je používaná na typoch striech, pre ktoré je určený náš systém má

dvadsaťšesť metrov štvorcových. Preto celkovú plochu len jednoducho vydelíme dvadsiatimi šiestimi a dostaneme potrebný počet balíkov. K celkovej ploche je ešte pred delením potrebné pripočítať desať percent, ako stratné v detailoch na streche. Backend aplikácie k danej cenovej ponuke priradí položku, ktorú vypočítala, jej množstvo a sumu, ktorá závisí od jednotkovej ceny, ktorá je definovaná v sklade.

- Potrebné skrutky na ukotvenie strešnej fólie. Nakoľko aplikácia má vypočítanú spodnú plochu strechy, pre ktorú je vytváraná CP a vieme, že na kotvenie je potrebné 6 skrutiek do jedného metra štvorcového, opäť za nás program spraví jednoduchý výpočet, pridá položku, jej množstvo a cenu do cenovej ponuky.
- Potrebné okapové plechy. Každá atika na streche je zakončená takzvanými okapovými plechmi. Tieto okapové plechy sa počítajú na pokrytie celkej dĺžky súčtu všetkých atkík. Každý okapový plech má štandardne dva a pol metra, preto náš systém vydelí celkový súčet všetkých dĺžok atík dĺžkou jedného okapového plechu, zistí tým počet potrebných okapových plechov.
- Potrebný počet rohových líšt. Po obvode celej strechy a komínov sú potrebné rohové lišty, ktoré slúžia na kotvenie fólie v rohoch. Rovnako na základe vstupných parametrov, ktoré zadal užívateľ pre danú strechu sa vypočíta potrebné počet a suma za všetky, ktorá sa získa z aktuálne zadanej sumy v sklade.

Následne, pre celkové dokončenie vytvorenia cenovej ponuky z databázy, ktorú môže firma poskytujúca strechárske služby odoslať zákazníkovi na ohodnotenie, musí systém vygenerovať .xls súbor. Tento súbor je generovaný pomocou knižnice apache.poi. Systém vo svojom balíku vlastní preddefinovanú prázdnu CP, ktorú následne napĺna prechádzaním cez jednotlivé položky CP, ktorá je vytiahnutá z databázy. Do samotnej vygenerovanej CP už strechár dopíše len sumu za prácu a môže ju zaslať zákazníkovi.

- **Služby pre obsluhu stavov cenovej ponuky-**Objekt cenová ponuka vlastní enum atribút s názvom status. Tieto statusy, sme si v našej architektúre dopredu navrhli ako INREVIEW ACCEPTED,DECLINED

  - NEW - čo znamená že CP bola vytvorená a uložená do databázy, avšak neposunutá ďalej v procese.
  - INREVIEW - CP bola odoslaná zákazníkovi na posúdenie. Materiál sa v sklade neodpočítava a z hľadiska backendovej časti je tento status využitý najmä k

možnej filtrácii medzi cenovými ponukami.

- ACCEPTED - status, ktorý v preklade znamená schválené napovedá, o akú CP sa bude jednať. V PUT metóde, ktorá vykonáva aktualizovanie cenovej ponuky prebieha odpočet položiek zo skladu. Spustí sa algoritmus, ktorý preiteruje cez všetky položky, ktoré sa nachádzajú v cenovej ponuke a počet potrebných položiek pre danú zákazku odpočíta zo skladu. Tu vznikol menší problém, na ktorý sme narazili počas implementácie a spočíval v tom, že je potrebné oznámiť strechárovi množstvo položiek, ktoré je potrebné dokúpiť na danú zákazku, pretože sa jej na sklade nevyskytuje potrebné množstvo a je potrebné doskladniť ju.
- DECLINED - ide o zamietnuté CP, ktoré zákazníkovi boli odovzdané no zákazníková spätná väzba na cenovú ponuku bola záporná a preto potrebujeme takéto CP odfiltrovať. Zostávajú však archivované v databáze pre prípadnú ďalšiu možnosť prepoužitia.

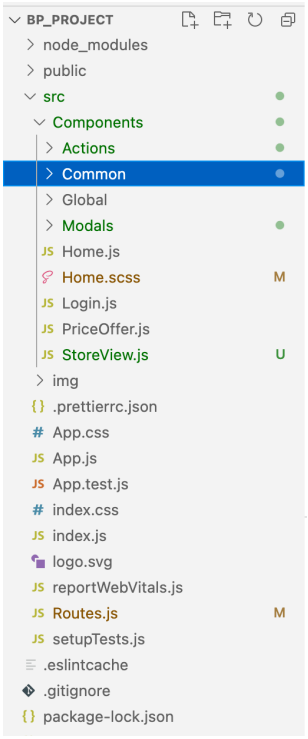
4.2 Frontendová časť

Po doimplementovaní všetkých potrebných častí backendovej časti a otestovaním funkčnosti a správnosti dát pomocou aplikácie Postman sme pokračovali implementáciou frontendovej časti. Dôležitou úlohou bolo premyslenie architektúry všetkých potrebných komponentov a využitie správnej knižnice na docielenie finálneho požadovaného dizajnu. V tejto časti práce si bližšie popíšeme štruktúru užívateľského rozhrania, jeho hlavné časti a postup, ktorý sme zvolili pri implementácii jednotlivých komponentov. Hneď v úvode implementovania sme sa rozhodli, že si architektúru projektu z hľadiska rozloženia komponentov nastavíme na pravú a ľavú časť obrazovky. Nakoľko sa jedná o tzv. single-page aplikáciu, rozhodli sme sa na ľavej strane renderovať komponenty, ktoré fungujú ako navigátor respektíve menu celej aplikácie. Všetky komponenty, ktoré sú potrebné pri vyrenderovaní akejkoľvek časti aplikácie sa nachádzajú tu. (menu, logo aplikácie, tlačidlo pre odhlásenie). Na pravej sa nachádza komponent, ktorý sme nazdvali screenRightSide a tvorí osemdesiat percent pravej strany monitora, ktorý užívateľ používa na zobrazenie aplikácie.

4.2.1 Štruktúra projektu

Na obrázku číslo 7 môžeme vidieť štruktúru projektu, ktorá sa skladá z nasledovných častí:

- Priečinok Components, v ktorom sa nachádzajú všetky komponenty, pomocou kto-



Obr. 8: Štruktúra frontendovej časti projektu

rých sme vyskladali jednotlivé časti našej aplikácie.

– **Folder Actions.** V tejto časti sú v projekte zadefinované všetky akcie, ktoré sa vykonávajú spolu s backendovou časťou. Všetko sú to funkčné komponenty, ktoré obsahujú metódy spracúvajúce dáta, ktoré je potrebné posielať na serverovú časť. Každá takáto akcia obsahuje premennú config, v ktorej sú zadefinované viaceré parametre, potrebné na odoslanie rest požiadavky na server. V tomto configu sa nachádza URI adresa, na ktorú sa daná požiadavka musí posielať, zadefinovaný typ tela požiadavky, zadefinované dáta, ktoré sa posielajú v tele požiadavky. Ďalej určujeme danej akcii o akú HTTP metódu sa jedná a parametre, ktoré sa posielajú spolu s requestom (v našom prípade token, kvôli autorizácii requestu, ktorý je uložený v local storage prehliadača). Nasleduje definicia axios, ktorá slúži na komunikáciu medzi externým zdrojom (našim backendom). V tejto definícii sa odošle požiadavka podľa nášho configu a po vykonaní tejto požiadavky sa vykonajú funkcie, ktoré zadefinujeme v akcii pre ďalšie spracovanie dát. V prípade, že nám server odpovedá statusom 200, ide o úspešnú požiadavku a my vieme, že môžeme pokračovať v aplikácii ďalej definovanými algoritmami. V prípade iných odpovedí môžeme užívateľovi oznámiť

popupom chybu, alebo vykonať iné žiadané správanie aplikácie.

```
export function getAllItems(onSuccess) {
  const config = {
    method: 'GET',
    url: 'http://localhost:8080/api/item/getAllItems',
    responseType: 'json',
    headers: {
      Authorization: 'Basic ' + btoa('mia-dms-gui:secret'),
      'Content-Type': 'application/json'
    },
    params: {
      access_token: localStorage.getItem('access_token')
    }
  };
  axios
    .request(config)
    .then(response => {
      onSuccess(response.data);
    })
    .catch(error => {
      if (error.response !== undefined) {
        if (error.response.status === 401) {
          console.log(error.response);
        }
        if (error.response.status === 403) {
```

Obr. 9: Ukážka kódu akcie vo frontendovej časti

- **Common.**V tomto priečinku sa nachádzajú komponenty, ktorými zabezbečujeme rozdelenie vačších celkov do menších častí, vďaka ktorým je kód priehľadnejší a má omnoho väčšie prepoužitie v prípade, že potrebujeme nejakú časť oplikácie renderovať na viacerých miestach. Pre každý takýto komponent je vytvorený zvlášť priečinok, v ktorom sa nachádza prvý súbor s príponou .js, čo znamená, že sa jedná o javascriptový súbor a druhý, s príponou .scss, čo znamená, že ide o štýl daného komponentu.
- **Global.** Na tomto mieste sú v projekte zadefinované globálne premenné ako napríklad url adresa na server, na ktorý chceme posielať naše požiadavky z užívateľského rozhrania. Ďalej tu je funkčný komonent zodpovedný za spracovávanie funkcionality, ktorá sleduje platnosť autentifikovaného tokenu, s ktorým sme na backende prihlásení a v prípade, že jeho platnosť vyprší, automaticky nás odlhási. Platnosť tokenu sa zisťuje porovnaním času, kedy bol token vygenerovaný, aktuálnym časom a dĺžky tokenu.
- **Modals.** Pod priečinkom modals sa nachádzajú všetky komponenty, ktoré fungujú ako takzvané modaly. Jedná sa o modálne okná, ktoré sú renderované nad hlavnou časťou našej aplikácie. My ich využívame najmä pri všetkých akciách, pri ktorých potrebujeme zobraziť nejaké prídavné dáta alebo chceme pridať nejaké dáta, ktoré musí používateľ zadať ako vstup. Pre tieto modaly, nakoľko sú

všetky veľmi podobné, sme sa rozhodli vytvoriť len jeden súbor, ktorý obsahuje všetky štýly týchto modalov a v prípade potreby pridať nejaké dodatočné sme riešili tento problém zadefinovaním štýlu pomocou vlastností daného modalu. Obsluhu modalov sme vyriešili premennou, ktorá je typu boolean a určuje, či modal je, alebo nie je otvorený. Ďalej sme naimplementovali metódu, ktorá túto hodnotu pri zavolaní zneguje a tým vieme vypnúť, zapnúť zobrazovanie modalu priamo na náš klik.

  - Ďalej máme v projekte súbory home.js, login.js, priceOffer.js a storeView.js, ktoré slúžia pri preklikávaní medzi jednotlivými položkami v menu. Podľa zakliknutia sa renderujú jednotlivé súbory. Rovnako ide o komponenty.

- V hlavnej zložke src sa nachádza súbor routes.js, ktorý je jeden z najdôležitejších komponentov, ktoré sú využívané pri prepínaní jednotlivých položiek v menu. V projekte sme si zadefinovali dva typy routov. Prvý je obyčajný route a druhý - privátny. Pomocou privátneho routingu vieme zabezbečiť, že v prípade, ak by útočník, ktorý nie je prihlásený, poznal url adresu, ktorá smeruje v našom prípade napríklad do skladu, či na hlavnú stránku, nemôže ju zobraziť bez toho, aby mal frontend v internetovom prehliadači uložený aktuálny prístupový token. V momente, ak nemá router k dispozícii aktuálny prístupový token, presmeruje používateľa na stránku, na ktorej sa môže opäť prihlásiť.

- V priečinku nodeModules sa nachádzajú všetky potrebné knižnice, ktoré sme do nášho projektu nainštelovali pomocou manažéra node balíkov pomocou príkazu npm install "názov balíka".

4.2.2 Implementačná časť prihlásenia a odhlásenia

Prihlásenie používateľa je prvý krok, ktorým sa môže autorizovať na vykonávanie ďalších požiadaviek z užívateľského rozhrania na serverovú časť aplikácie. Počas tejto akcie sa posiela na serverovú časť požiadavka (HTTP metóda POST), ktorú nie je potrebné autorizovať tokenom. Do jej tela posielame ako parametres prihlasovacie meno a heslo.

V obrázku číslo deväť môžeme vidieť časť kódu, ktorá sa vykoná po nastavení konfiguárcie, v ktorom určíme URI adresu (v našom prípade oauth/token) a telo požiadavky. S nasledovnou konfiguráciou sa odošle požiadavka na server. Tam sa pomocou Oauth2 autentifikácie overí zadané meno a heslo a v prípade správnosti údajov príde v odpovedi zo servera:

- expiresIn - atribút, ktorý určuje dobu, po ktorej vyprší prístupový token do aplikácie. Táto hodnota je vrátená v milisekundách.

```
.request(config)
.then((response) => {
    Date.prototype.addSeconds = function (seconds) {
        this.setSeconds(this.getSeconds() + seconds);
        return this;
    };

    const expires_in = new Date().addSeconds(response.data.expires_in);
    localStorage.setItem("expires_in", expires_in.getTime());
    localStorage.setItem("access_token", response.data.access_token);
    localStorage.setItem("refresh_token", response.data.refresh_token);
    localStorage.setItem("token_type", response.data.token_type);
    localStorage.setItem("user_name", userName);
    localStorage.setItem("is_admin", "");
    props.history.push('/')
})
.catch((err) => {
    if (err.response != null) {
        if (err.response.data != null) {
            if (err.response.data.error === "invalid_grant") {
            }
```

Obr. 10: Ukážka časti kódu, v ktorej prebieha login na strane frontendu.

- access-token - hodnota, ktorá predstavuje dvadsaťšesť miestny kód zložený z číslic, písmen a štyroch pomlčiek, pomocou ktorého sa autorizujú požiadavky poslané na server.

- refresh-token - vygenerovaný obnovujúci token, ktorý slúži v aplikácii ako predpríprava na ďalšie vylepšenia. Pomocou obnovujúceho tokenu bude možné v ďalšej verzii aplikácie obnoviť prístupový token po jeho vypršaní. Táto funkcionalita sa využíva najmä pri kontrolovanom odhlásení užívateľa po dlhšej dobe nečinnosti (čas, ktorý predstavuje doba platnosti prístupového resp. obnovujúceho tokenu).

- Token type - určuje typ tokenu, ktorý je vygenerovaný a jeho použitie. V našom prípade sa generuje typ tokenu bearer, čo znamená, že je určený na autorizáciu požiadaviek.

- Ďalej sa vracia užívateľské meno aktuálne prihláseného užívateľa a boolean hodnota isAdmin, ktorá určuje, či má užívateľ administrátorské práva, alebo nie. Opäť sa jedná o predprípravu do ďalších verzii aplikácie, kedy by bolo možné v budúcnosti rozdeľovať práva jednotlivých používateľov.

Všetky tieto hodnoty sa ukladajú do takzvaného local storage-u. Jedná sa o úložisko v internetovom prehliadači poúživateľa. Toto úložisko sme sa rozhodli použiť, pretože

je vhodné používať ho na ukladanie dát, ktoré sú konštantné počas celého chodu našej aplikácie.

Po ukončení používania aplikácie je vhodné, ak sa používateľ odhlási a nebude tým riskovať zneužitie dát alebo celkovo prihláseného užívateľského účtu. Toto odhlásenie prebieha v dvoch častiach. V prvej časti sa po kliknutí v užívateľskom rozhraní na tlačidlo ödhlásiť"zavolá požiadavka (HTTP metóda DELETE), ktorá na serverovej časti vymaže aktuálne vygenerovaný platný prístupový a obnovujúci token. Následne v prípade, že serverová časť vráti status 200, čo znamená, že požiadavka sa vykonala správne, príde vykonanie funkcie, ktorá vyčistí úložisko prehliadača, v ktorom máme uložené informácie z posledného prihlásenia a je potrebné ich zmazať aj lokálne. Nakoľko je náš router nakonfigurovaný s privátnym routingom, v prípade, že sa v prehliadačovom úložisku nenachádza platný prístupový token, bude používateľ po kliknutí na odhlásenie a vykonaní dvoch predošlých krokov presmerovaný na stránku prihlásenia. Tu treba poznamenať, že po opätovnom prihlásení server vygeneruje a vráti opäť nové tokeny a opäť ich uloží aj na lokálnej strane užívateľa.

4.2.3 Implementačná časť vytvorenia cenovej ponuky

Hlavnou časťou našej aplikácie je vytvorenie cenovej ponuky. Užívateľovi sa po zvolení položky "naceniť strechu", ktorá sa nachádza v menu, vyrenderuje na obrazovku niekoľko vstupných polí, ktoré musí vyplniť a potvrdzujúcim tlačidlom. Jedná sa o formulár, pri ktorom používateľ zadávaním vstupných hodnôt mení inicializované premenné v takzvanom state. Tieto premenné sa menia v každom vstupnom poli tým, že reagujú na akciu "zmenaä tým vieme zabezpečiť, že sú stále aktuálne.

Z premenných, ktoré máme uložené aktuálne počas vytvárania cenovej ponuky zostavíme cenovú ponuku poslaním dvoch požiadaviek na server. V prvej požiadavke (HTTP metóda POST) sa vytvorí v databáze objekt strecha, ktorej serverová časť aplikácie vypočíta všetky potrebné položky a k tejto streche sa ďalej vytvorí prázdna cenová ponuka. Následne keď dostaneme v odpovedi zo servera id tejto cenovej ponuky, je odoslaná z frontendovej časti (automaticky, nemusí ju posielať používateľ žiadnou akciou) ďalšia požiadavka (HTTP metóda PUT), ktorá vyvolá na serverovej časti proces tvorenia cenovej ponuky a vytvorenia excel súboru.

4.2.4 Implementačná časť správa skladu

Po dokončení implementácie vytvorenia cenovej ponuky sme začali s implementáciou skladu. Prvou časťou, ktorú bolo dôležité premyslieť bolo správne usporiadanie jednotlivých komponentov na obrazovke kvôli priehľadnosti a celkovému dizajnu. Rozhodli sme

```
)
class NewPriceOfferForm extends Component {
  constructor(props) {
    super(props);
    this.state = {
      heigth: 0,
      width: 0,
      isOpenAddAttic: false,
      isOpenAddChimney: false,
      customerName: '',
      roofId: '',
      atiky: [],
      chimneys: []
    };
  }
;
```

Obr. 11: State v komponente pre tvorbu novej cenovej ponuky

sa všetky jednotlivé položky v sklade vykreslovať v komponente pripomínajúcom tabuľku, v ktorej každý riadok bude reagovať na udalosť kliknutie a bude možné zobrazovať následne modal, v ktorom sa budú upravovať atribúty danej položky. Rozhodovali sme sa medzi použitím knižnice Reactstrap a material-ui. Obe spomenuté dostatočne pokrývali požiadavky, ktoré sme mali (prepoužiteľnosť, jednoduchosť, udalosť na kliknutie), avšak kvôli pomerne dobrej dokumentácii, ktorou material-ui disponuje sme sa rozhodli zvoliť práve ten a využiť jeden z jeho komponentov s názvom data-grid. Na vykreslenie takéhoto datagridu je potrebné preddefinovať počet stĺpcov tabuľky, ich mená, šírku, či sú skryté alebo viditeľné a ich id hodnotu. Ďalej nastavíme veľkosť jednej stránky - počet riadkov vykreslených na jednej strane. Následne sme z backendovej časti použili api, ktoré nám v odpovedi vrátilo všetky dostupné položky na sklade a túto odpoveď zapísali do premennej v state. Výhodou komponentu, ktorý sme použili je, že má funkcionalitu "samomapovania". To znamená, že ak do počas vytvárania tabuľky pošleme do premennej, ktorá zodpovedá riadkom našej tabuľky pole objektov, ktoré majú atribúty s názvami, ktoré sa zhodujú s názvami stĺpcov, automaticky je tento objekt namapovaný. Po každej zmene položiek v state sa volá funkcia, ktorá nanovo prerenderuje obrazovku s novými dátami. Vždy, po každej zmene, ktorú spravíme na frontendovej časti voláme funckiu, ktorá získa zo servera položky nachádzajúce sa v databáze s ich atribútmi. Týmto vieme zabezpečiť, že zobrazované data sú stále aktuálne. Po kliknutí na jeden z takto vyrenderovaných riadkov sa zobrazí modal, v ktorom vieme zmeniť množstvo danej položky na sklade, jej veľkosť a cenu. Následne po zadaní zmenených atribútov je možné kliknúť

na tlačidlo äktualizovať", ktoré pošle na server požiadavku (HTTP metóda PUT) a v databáze server aktualizuje prijaté dáta. Následne prebieha script, ktorý obsahuje volanie požiadavky na server, pre aktualizovanie položiek, získaním všetkých z databázy.

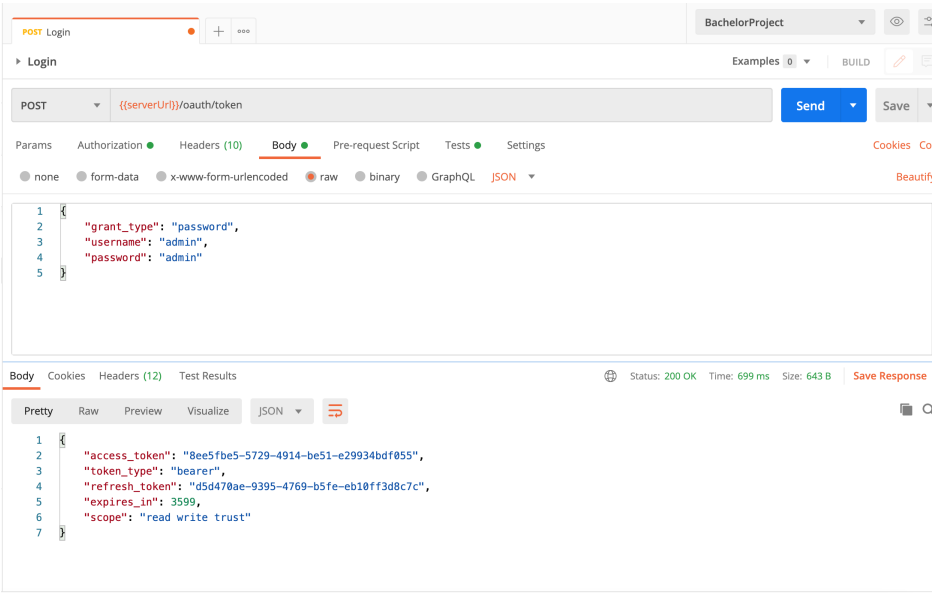
4.2.5 Implementačná časť aktualizácia cenovej ponuky

Ako poslednú časť na frontendovej časti sme venovali implementácii samotnej aktualizácie cenovej ponuky a spracovania procesu. V našej prvotnej verzii aplikácie boli vykreslené na úvodnej obrazovke všetky cenové ponuky bez rozdielu na ich status. Rozhodli sme sa tento náhľad zmeniť pomocou pridania navigačných tlačidiel nad tabuľku, v ktorej sú vykreslené všetky cenové ponuky. Reactstrap disponuje veľmi kvalitným komponentom, v ktorom si vie programátor zadefinovať všetky taby, ktoré bude navigačný panel ponúkať. Následne je potrebné zadefinovať obsah všetkých položiek. Tento komponent sleduje aktívny číslo uložené v premennej, ktorú do tohoto komponentu posielame pomocou props. V prípade, že chceme zmeniť obsah, ktorý je vyrenderovaný na obrazovku, kliknutím na jednu z položiek v navigačnom paneli sa vyvolá udalosť, ktorá počúva na kliknutie. V tejto udalosti je zavolaná funkcia, ktorá v state zmení premennú äctiveTab"na iné číslo a tým, že sa zmení state sa prerenderuje stránka nanovo, ale už s novými dátami. Tento navigačný panel funguje na filtrovanie cenových ponúk podľa ich stavu, v ktorom sa aktuálne nachádzajú. Napríklad po kliknutí na položku "nové"v navigačnom paneli sa prerenderuje obrazovka nanovo. Z frontendovej časti sa pošle požiadavka na server (HTTP metóda GET), ktorá vráti v odpovedi všetky cenové ponuky so statusom nová.

4.3 Testovanie aplikácie

Veľmi dôležitou časťou, ktorá patrí ku implementácii každého systému je overenie všetkých funkčných požiadaviek. Takéto testovanie sme vykonávali počas implementácie dva krát. Prvý krát po dokončení serverovej časti aplikácie, kedy sme všetky požiadavky, ktoré sme mali v pláne používať na frontendovej časti, otestovali pomocou aplikácie postman. Počas testovania sme si v postmanovi vytvorili prostredie, do ktorého sme si naukladali všetky potrebné požiadavky s uri. Následne sme do týchto požiadaviek pripravili telo požiadavky, ktoré sme posielali ako JSON. Vo všetkých scenároch sme potrebovali zistiť, či nie je pri niektorej z požiadaviek na server odpoveď päťsto, respektíve iná ako dvestovková alebo štyristovková. Odpoveď päťsto pre nás znamená chybu v backendovej časti, ktorá nie je očakávaná ani nijakým spôsobom ošetrená. Začali sme samozrejme testovaním prihlásenia, nakoľko to je prvá akcia v systéme, ktorá je nadradená všetkým ostatným úkonom našej aplikácie. Testované boli nasledovné scenáre: zadanie zlého užívateľského mena, zadanie zlého užívateľského hesla a zadanie správnych údajov. V prípade,

že sme zadali všetko správne, server vrátil status 200 a v odpovedi údaje o vygenerovanom tokene.

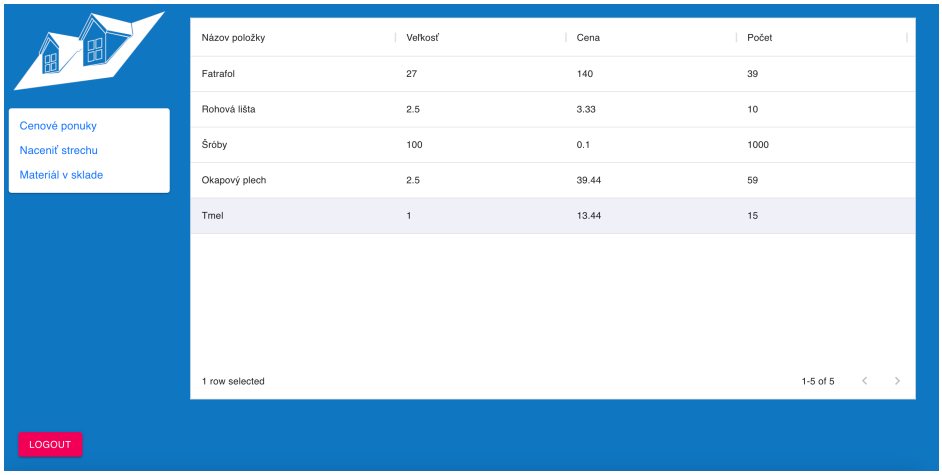


Obr. 12: Testovanie prihlásenia v aplikácii postman

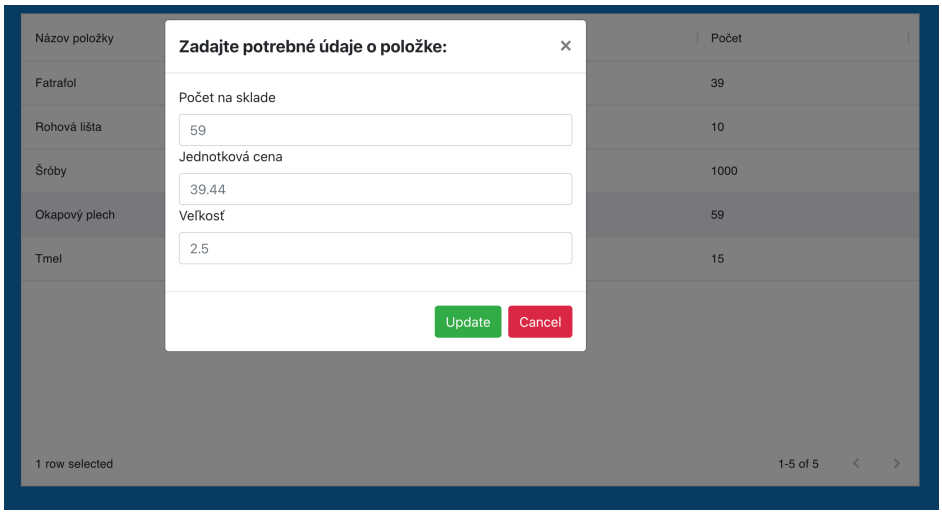
Tak, ako vidíme na obrázku číslo 12, v prípade, že sme zadali všetko správne, server vrátil status 200 a v odpovedi údaje o vygenerovanom tokene. V prípade, že sme poslali na server zlé prihlasovacie meno, či heslo, server vyhodnotil autentifikáciu za neúspešnú a odpovedal odpoveďou so statusom 400. Okrem tohoto statusu prišla v odpovedi aj chybová hláška ïnvalid grantä popis chybovej hlášky "bad credentials", čo znamená, že boli zadané zlé prihlasovacie údaje. Rovnaká odpoveď nám prišla aj v prípade, že sme meno, alebo heslo vôbec neposlali. Ďalej sme pokračovali otestovaním všetkých POST metód, ktoré nám slúžili na naplnenie databázy potrebnými objektami. Všetky tieto metódy fungovali v poriadku a server nám vždy odpovedal statusom 200. Týmto spôsobom sme aj otestovali všetky HTTP GET metódy, kedy sme najprv otestovali všetky, ktoré nám v response vracali všetky údaje, ktoré nevyžadovali žiaden parameter ako vstup. Všetky tieto testy, až na jeden, sa nám podarilo spustiť úspešne a server nám vždy odpovedal statusom 200. V jednom prípade, kedy sme sa dopytovali po objektoch, ktoré obsahovali obosjmerné namapovanie tabuliek nastávala cyklická referencia. Presnejšie, išlo o dopyt na užívateľa, ktorý vlastnil atribút autorita. Táto autorita opäť obsahovala referenciu na užívateľa, ktorý obsahoval referenciu na túto istú autoritu a server to vyhodnotil ako chybu a vrátil status 500. Našim riešením bolo použitie anotácie @JsonIgnore nad atribút users v objekte autorita, čo znamenalo, že v odpovedi bude tento atribút ignorovaný a tým bude cyklická referencia prerušená. Následne po tejto oprave prebehli všetky GET

metódy korektne a vrátili status 200. Pokračovali sme http metódamy PUT, pretože ich využitie v našom systéme je veľmi dôležité kvôli aktualizovaniu dát v databáze. Všetky tak, ako sme predpokladali fungovali správne a http metódy DELETE tiež.

Pokračovali sme testovaním jednotlivých funkčných požiadaviek z frontendovej časti. Začali sme opäť testovaním prihlásenia. Na frontendovej časti tak, ako už bolo spomenuté, máme router rozdelený na privátnu a verejnú časť. Ku privátnym routom nemôže byť možné pristúpiť bez prihlásenia. Preto sme sa otestovali url "localhost:4403/home". Táto url adresa je využívaná na zobrazenie domovskej stránky, pričom ide o privátnu adresu. Router je nastavený tak, aby boli všetky url cesty presmerované na prihlasovaciu stránku a tieto testy dopadli úspešne. Ďalej sme pokračovali otestovaním správy skladu. Ide o sekundárnu funkcionalitu systému, avšak je nevyhnutná pre vytvorenie cenovej ponuky. Cenová ponuka je vyskladaná zo základných položiek, ktorých počet systém generuje na základe vstupných údajov a na zistenie ich ceny je potrebné mať v sklade definovanú ich cenu za jeden kus. Preto sme testovali fuknčnosť renderovania komponentov po kliknutí na položku z menu "Materiál v sklade ". Očakávali sme vyrenderovanie tabuľky, ktorá bude obsahovať stĺpce názov položky, veľkosť, cena a počet kusov. Každý z týchto riadkov musí byť klikateľný, čo znamená, že po kliknutí na jeden z riadkov sa musí vyvolať požadovaná akcia, ktorú sme definovali. V našom prípade ide o rozkliknutie modálneho okna, v ktorom dokážeme aktualizovať položku, ktorá sa nachádzalo v riadku, na ktorý sme klikli.



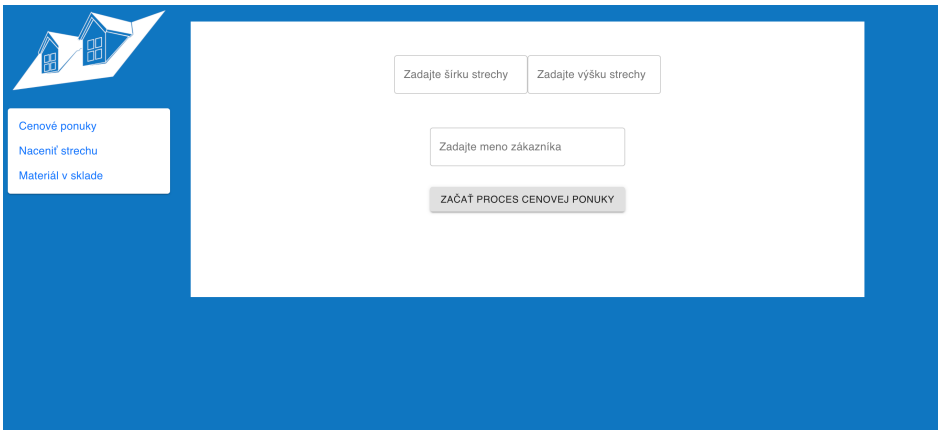
Obr. 13: Obrazovka po kliknutí na položku "Materiál v sklade"



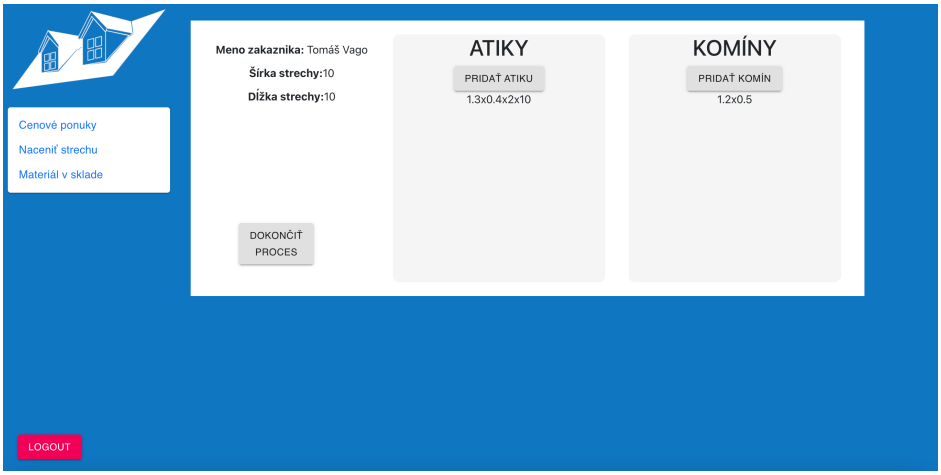
Obr. 14: Modálne okno pre aktualizáciu položky

Tak, ako môžeme vidieť na obrázku číslo štrnásť, po kliknutí na položku okapový plech vo vyrenderovanej tabuľke sa zobrazilo modálne okno, v ktorom sa nachádzajú tri vstupné polia, do ktorých zadáme vstupné parametre o danej položke. Po kliknutí na tlačidlo Update sa aktualizoval zobrazovaný zoznam položiek, pričom údaje, ktoré sme zmenili v modálnom okne boli už aktuálne.

Po potvrdení korektnej funckionality správy skladu sme pokračovali na samotný proces vytvorenia cenovej ponuky. Preklikávania položiek v menu fungovalo správne, takže bolo možné pokračovať zadaním vstupných parametrov strechy. Nakoľko je celý systém nastavený na počítanie v metroch, vstupné údaje sa zadávajú v metroch. Po zadaní šírky, dĺžky strechy a mena zákazníka do vstupných polí sa správne po kliknutí na tlačidlo "začať proces cenovej ponukyödoslala požiadavka na serverovú časť, kde sa do databázy uložila prázdna strecha, ktorá ešte nemala vypočítané potrebné položky. Táto strecha je v databáze evidovaná ako prázdna, má priradenú len veľkosť.



Obr. 15: Obrazovka pred začatím procesu tvorenia cenovej ponuky"



Obr. 16: Obrazovka počas procesu tvorenia cenovej ponuky

Počas vytvárania sme pridali streche jeden komín a jednu atiku (táto funkcionalita je veľmi dôležitá, pretože strechy môžu alebo nemusia mať tieto položky a je nevyhnutné v systéme mať možnosť dynamicky ich pridávať). Na obrázku č.16 vidíme naľavo vypísané rozmery strechy a meno zákazníka a vpravo v dvoch oknách zvlášť vypísané rozmery pridaných špecifických položiek. Po dokončení tohoto procesu bol vygenerovaný xls súbor v priečinku, ktorý je preddefinovaný v konfigurácii systému.

STRECHY vAGO S.R.O

- MONTÁŽ PLOCHÝCH A ŠIKMÝCH STRIECH
-MONTÁŽ ALTÁNKOV
-IZOLÁCIE SPODNÝCH STAVIEB

STRECHY VAGO S.R.O
KLASOV 242, 951 53
TEL.Č. +421 915 419 493
E-MAIL: ALFARESCUE@GMAIL.COM

Cenová Ponuka :

odberateľ:	Tomáš Vago		
stavba:			
adresa:			
telefón:			
fax, e-mail:			
odoslaná:			
platnosť:	30 dní		
vybavuje:	Vago		

Názov	Pocet	Euro s DPH
Materiál		
Fatrafol	6	840,00 €
Okapový plech	12	473,28 €
Rohová lišta	6	19,98 €
Šróby	600	60,00 €
Práca		

Strana 1

Obr. 17: Vygenerovaná cenová ponuka

Preddefinované položky sa vypočítali pre danú strechu, pridali sa do správnych buniek vo vzore cenovej ponuky a rovnako k nim sa pridelila cena. Do takejto cenovej ponuky je potrebné už pridať len cenu za prácu a je možné ju poslať zákazníkovi na posúdenie. Po odoslaní na posúdenie zákazníkovi vyžadujeme od systému zmenu statusu cenovej ponuky. Táto funckionalita fungovala správne, status v modálnom okne bolo možné zmeniť. Po kliknutí na tlačidlo, ktoré mení status cenovej ponuky na potrebnú hodnotu sa odosiela z frontendovej časti PUT HTTP metóda, ktorá uloží posielaný status do databázy. Následne sa obnoví zoznam s aktuálnymi dátami.

Záver

Hlavným cieľom tejto bakalárskej práce bolo zhotovenie webovej aplikácie, ktorá zamestnancom strechárskej firmy, uľahčí jeho prácu a zníži riziko chyby. Počas vytvárania novej cenovej ponuky musí strechár počítať veľké množstvo jednoduchých malých vzorcov a následne tieto výsledné hodnoty sčítavať dokopy. Počas sezóny strechár nacení nespočetné množstvo striech a naša implementovaná aplikácia dokáže ušetriť pri každej cenovej ponuke niekoľko minút.Počas roka zamestnanci vytvoria veľké množstvo cenových ponúk, čo znamená veľké ušetrenie času vďaka implementovanej webovej aplikácii a táto okolnosť je hlavný prínos našej práce pre spoločnosť, respektíve pre pracovníkov v strechárskej firme.

Celá architektúra aplikácie je pripravená tak, aby bolo možné v budúcnosti implementovať rôzne doplnky a nápady, ktoré môžu proces nacenenia cenovej ponuky uľahčiť strechárovi a tým napomôcť k spokojnosti zákazníka, ktorý dostane vypracovanú cenovú ponuku rýchlejšie.

Počas práce na aplikácii sa nám úspešne podarilo splniť všetky body, ktoré boli požadované na bakalársku prácu a za použitia REST princípov sme vytvorili webovú aplikáciu, ktorá využíva mySQL databázu na ukladanie všetkých dát. Pre backendovú časť sme vytvorili samostatnú časť, ktorú je možné nasadiť napríklad na tomcat a využívať všetky jeho api. Dokumentácia jednotlivých REST API sa nachádza v prílohe C. Ako programovací jazyk sme použili Java a framework Spring, v ktorom sme získali nové skúsenosti a vedomosti. Frontendovú časť sme implementovali vo frameworku React, ktorý sa ukázal ako správna voľba, pretože disponuje dostatočne veľkým množstvom knižníc, ktoré sa dajú využiť na zjednoduchšenie programátorskej práce. Všetky jednotlivé funkcionality boli úspešne otestované a spĺňajú všetky požiadavky. Rovnako s úspešným záverom sa nám podarila funkcionalita správy skladových zásob. Ďalším bodom v požiadavkách našej práce bolo zhotovenie užívateľskej príručky, ktorá je priložená v prílohách ako príloha A.

Zoznam použitej literatúry

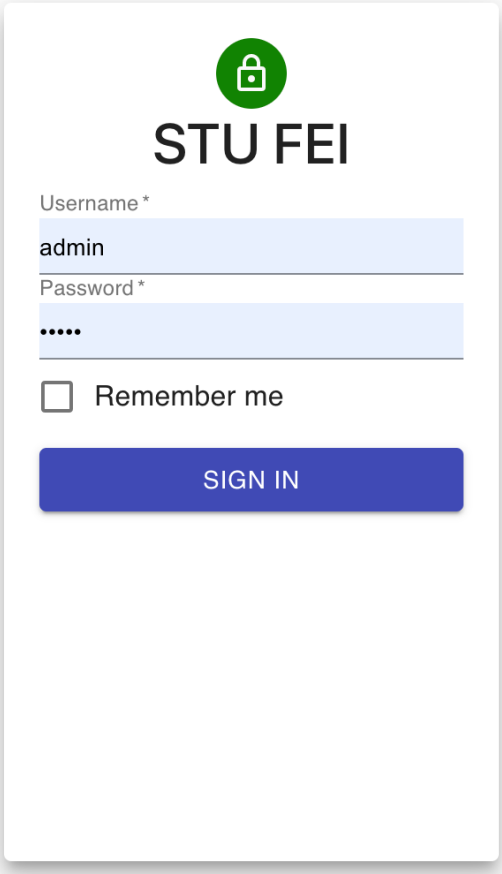
1. KHAN, Wisal, AHMAD, Waqas, LUO, Bin a AHMED, Ejaz. SQL Database with physical database tuning technique and NoSQL graph database comparisons. In: *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. 2019, s. 110–116. Dostupné z DOI: 10.1109/ITNEC.2019.8729264.
2. CHRISTENSSON, Per. *"Framework Definition.* [https://techterms.com/definition/framework]. 2013. [Online; Accessed May 27, 2021].
3. TILKOV, Stefan a VINOSKI, Steve. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*. 2010, roč. 14, č. 6, s. 80–83. Dostupné z DOI: 10.1109/MIC.2010.145.
4. GUNTUPALLY, Kavya, DEVARAKONDA, Ranjeet a KEHOE, Kenneth. Spring Boot based REST API to Improve Data Quality Report Generation for Big Scientific Data: ARM Data Center Example. In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, s. 5328–5329. Dostupné z DOI: 10.1109/BigData.2018.8621924.
5. GUPTA, Guatam. *What are the disadvantages of the Spring Framework?* [https://www.quora.com/What-are-the-disadvantages-of-the-Spring-Framework]. 2017. [Online; accessed 28-January-2017].

Prílohy

A Používateľská príručka

A.1 Prihlásenie do systému

Na prihlásenie do systému musí užívateľ zadať užívateľské meno a heslo, ktoré bolo vygenerované systémom počas inicializácie aplikácie, alebo správcom aplikácie, ktorý dodatočne pridal používateľa do databázy.

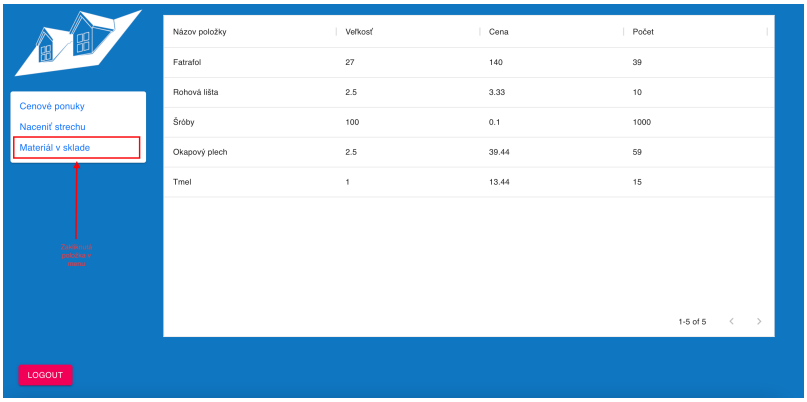


Obr. A.1: Obrazovka prihlásenia

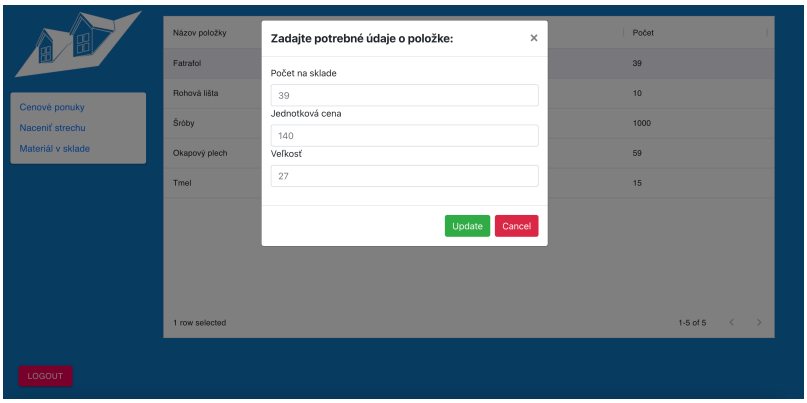
A.2 Aktualizácia skladu

Pre aktualizáciu skladu musí užívateľ kliknúť na položku v menu, ktoré sa nachádza na ľavej strane obrazovky. Následne sa mu zobrazí tabuľka, v ktorej vidí zoznam položiek, ktoré sa nachádzajú na sklade. Pre aktualizovanie jednotlivých atribútov ľubovolnej položky v sklade je potrebné kliknúť priamo do riadu, v ktorom sa daná položka nachádza. Následne sa zobrazí modálne okno, do ktorého je možné vpisovať hodnoty, ktoré chce

užívateľ meniť. Po nastavení všetkých hodnôt je potrebné kliknúť na tlačidlo update.



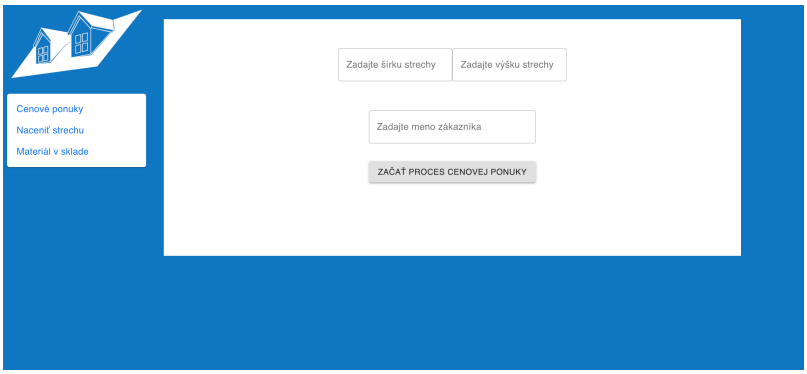
Obr. A.2: Položka v menu, na ktorú je potrebné kliknúť



Obr. A.3: Modálne okno, v ktorom je možné aktualizovať atribúty položky.

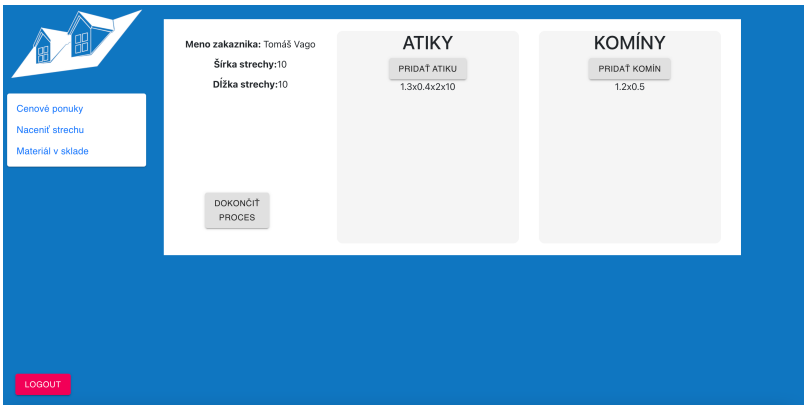
A.3 Proces vytvorenia cenovej ponuky

Začatie procesu cenovej ponuky sa vykonáva v položke menu "Nacenit ponuku".



Obr. A.4: Prvý formulár pri procese novej cenovej ponuky

V prvom formulári musí užívateľ zadať šírku a dĺžku strechy. V spodnom vstupnom poli musí zadať meno zákazníka. Následne musí kliknúť na tlačidlo s nápisom "Začať proces cenovej ponuky".



Obr. A.5: Druhý formulár pri procese novej cenovej ponuky

Po presmerovaní užívateľa na druhý formulár je potrebné pridať na strechu komíny a atiky. Tie sa pridávajú na strechu pomocou tlačidiel pridať atiku alebo pridať komín. Po zobrazení modálneho okna musí užívateľ pre každú túto špecifickú položku zadať jej rozmery a kliknúť na tlačidlo upload. Po pridaní všetkých položiek je potrebné kliknúť na tlačidlo "dokončiť proces". Po presmerovaní na prvý formulár bola cenová ponuka úspešne vytvorená a jej vygenerovaný excel súbor nájdeme v adresári, ktorý je špecifikovaný v configurácii aplikácie.


STRECHY VAGO S.R.O
- MONTÁŽ PLOCHÝCH A ŠIKMÝCH STRIECH
-MONTÁŽ ALTÁNKOV
-IZOLÁCIE SPODNÝCH STAVIEB

Cenová Ponuka :

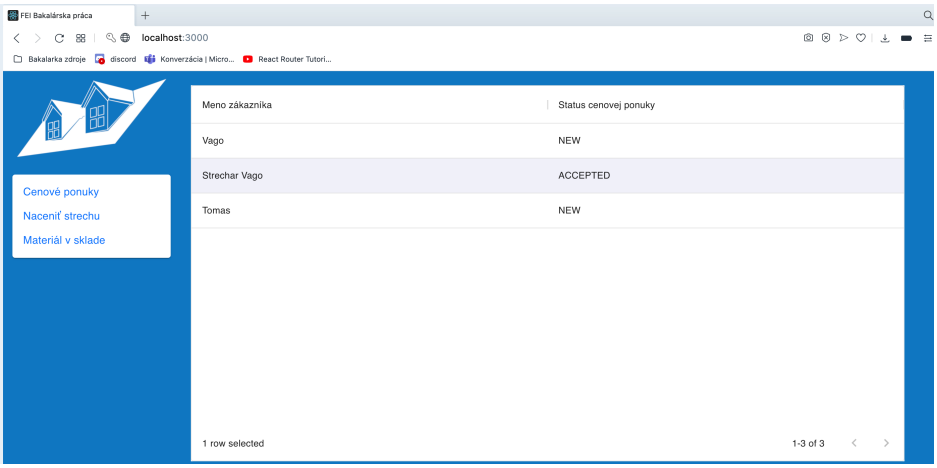
odberateľ:	Tomáš Vago
stavba:	
adresa:	
telefón:	
fax, e-mail:	
odoslaná:	
platnosť:	30 dní
vybavuje:	Vago

Názov	Počet	Euro s DPH
Materiál		
Fatrafol	6	840,00 €
Okapový plech	12	473,28 €
Rohová lišta	6	19,98 €
Šróby	600	60,00 €
Práca		

Obr. A.6: Vygenerovaná cenová ponuka uložená do excel súboru

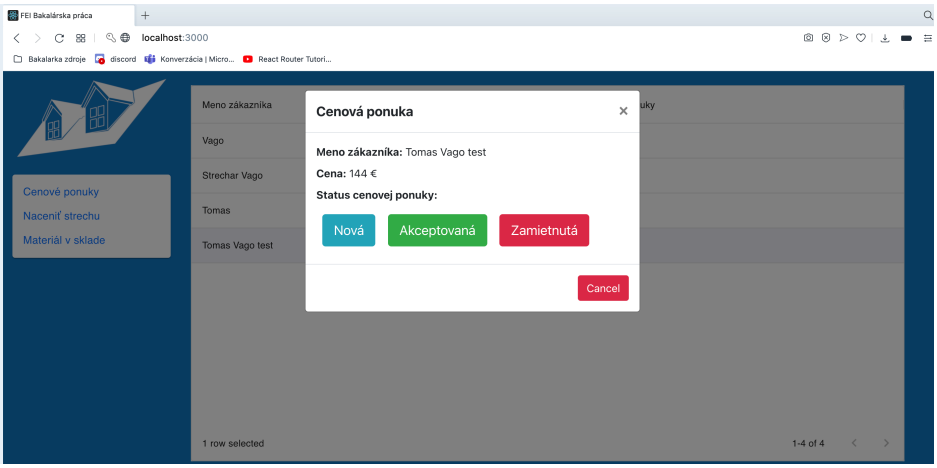
A.4 Aktualizácia statusu cenovej ponuky

Všetky cenové ponuky, ktoré sa nachádzajú v systéme nájde užívateľ v položke meno cenové ponuky, ktorá slúži zároveň aj ako domovská obrazovka.



Obr. A.7: Zoznam cenových ponúk

V prípade, ak chce užívateľ zmeniť status cenovej ponuky, musí kliknúť na riadok, v ktorom sa nachádza cenová ponuka a následne sa mu zobrazí modálne okno.



Obr. A.8: Modálne okno

Po kliknutí na jedno z farebne zvýraznených tlačidiel sa automaticky zmení status cenovej ponuky.

B Zdrojový kód aplikácie

Celý zdrojový kód aplikácie je dostupný na githube. Link, na ktorom je možné zdrojový kód webovej aplikácie naklonovať: https://github.com/TatkoSmollko/bachelorProject.git

C Dokumentácia ku vytvorenej API

Api dokumentácia je vygenerovaná pomocou developerského programu IntelliJ IDEA.

```yaml
openapi: "3.0.3"
info:
  title: "bachelor thesis API"
  description: "bachelor thesis API"
  version: "1.0.0"
servers:
  - url: "https://bachelor thesis"
paths:
  api/attics/create:
    post:
      summary: "POST api/attics/create"
      operationId: "createAttic"
      responses:
        200:
          description: "Api slúžiace na vytvorenie novej atiky."
  api/attics/deleteAttic/{id}:
    delete:
      summary: "DELETE api/attics/deleteAttic/{id}"
      operationId: "deleteAttic"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na vymazanie atiky s id posielaným v
requeste"
  api/attics/getAllAtticsByRoofId/{id}:
    get:
      summary: "GET api/attics/getAllAtticsByRoofId/{id}"
      operationId: "getAllAtticsByRoofId"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na získanie všetkých atík pre danú
strechu"
  api/attics/getAtticById/{id}:
    get:
      summary: "GET api/attics/getAtticById/{id}"
      operationId: "getAtticById"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
```

Obr. C.1: 1

```
          description: "Api slúžiace na získanie atiky z databázy na základe
jej ID"
  api/attics/updateAttic/{id}:
    put:
      summary: "PUT api/attics/updateAttic/{id}"
      operationId: "updateAttic"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na úpravu už existujúcej atiky."
  api/chimney/createChimney:
    post:
      summary: "POST api/chimney/createChimney"
      operationId: "createChimney"
      responses:
        200:
          description: "Api slúžiace na vytvorenie nového komína."
  api/chimney/deleteChimney/{id}:
    delete:
      summary: "DELETE api/chimney/deleteChimney/{id}"
      operationId: "deleteChimney"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na vymazanie komína."
  api/chimney/getAllChimneys:
    get:
      summary: "GET api/chimney/getAllChimneys"
      operationId: "getAllChimneys"
      responses:
        200:
          description: "Api slúžiace na získanie všetkých komínov v
databáze."
  api/chimney/getAllChimneysByRoofId/{id}:
    get:
      summary: "GET api/chimney/getAllChimneysByRoofId/{id}"
      operationId: "getAllChimneysByRoofId"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na získanie všetkých komínov podla id
strechy."
```

Obr. C.2: 2

```
  api/chimney/getChimneyById/{id}:
    get:
      summary: "GET api/chimney/getChimneyById/{id}"
      operationId: "getChimneyById"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na získanie komínu z databázy na základe
posielaného ID"
  api/item/createItem:
    post:
      summary: "POST api/item/createItem"
      operationId: "createItem"
      responses:
        200:
          description: "Api slúžiace na vytvorenie novej položky."
  api/item/deleteItem/{id}:
    delete:
      summary: "DELETE api/item/deleteItem/{id}"
      operationId: "deleteItem"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na vymazanie položky na základe ID."
  api/item/getAllItems:
    get:
      summary: "GET api/item/getAllItems"
      operationId: "getAllItems"
      responses:
        200:
          description: "Api slúžiace na získanie všetkých položiek."
  api/item/getByStore/{storeId}:
    get:
      summary: "GET api/item/getByStore/{storeId}"
      operationId: "getAllItemsByStoreId"
      parameters:
        - name: "storeId"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na získanie všetkých položiek v sklade
podla id skladu. "
  api/item/getItemById/{id}:
    get:
```

Obr. C.3: 3

```
      summary: "GET api/item/getItemById/{id}"
      operationId: "getItemById"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na získanie položky na základe id. "
  api/item/updateItem/{id}:
    put:
      summary: "PUT api/item/updateItem/{id}"
      operationId: "updateItem"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na úpravu existujúcej položky."
  api/priceOffer/createPriceOffer/{roofId}:
    post:
      summary: "POST api/priceOffer/createPriceOffer/{roofId}"
      operationId: "createPriceOffer"
      parameters:
        - name: "roofId"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na vytvorenie novej cenovej ponuky."
  api/priceOffer/deletePriceOffer/{id}:
    delete:
      summary: "DELETE api/priceOffer/deletePriceOffer/{id}"
      operationId: "deletePriceOffer"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na vymazanie cenovej ponuky."
  api/priceOffer/finishPriceOffer/{customer}/{priceOfferId}:
    get:
      summary: "GET
api/priceOffer/finishPriceOffer/{customer}/{priceOfferId}"
      operationId: "finishPriceOffer"
      parameters:
        - name: "customer"
```

Obr. C.4: 4

```
        in: "path"
        required: true
        schema:
          type: "string"
      - name: "priceOfferId"
        in: "path"
        required: true
        schema:
          type: "string"
    responses:
      200:
        description: "Api slúžiace na získanie dokončenej cenovej ponuky."
api/priceOffer/getAllPriceOffers:
  get:
    summary: "GET api/priceOffer/getAllPriceOffers"
    operationId: "getAllPriceOffers"
    responses:
      200:
        description: "Api slúžiace na získanie všetkých cenových ponúk."
api/priceOffer/getPriceOfferById/{id}:
  get:
    summary: "GET api/priceOffer/getPriceOfferById/{id}"
    operationId: "getPriceOfferById"
    parameters:
      - name: "id"
        in: "path"
        required: true
        schema:
          type: "string"
    responses:
      200:
        description: "Api slúžiace na získanie cenovej ponuky podľa ID."
api/priceOffer/updatePriceOffer/{id}:
  put:
    summary: "PUT api/priceOffer/updatePriceOffer/{id}"
    operationId: "updatePriceOffer"
    parameters:
      - name: "id"
        in: "path"
        required: true
        schema:
          type: "string"
    responses:
      200:
        description: "Api slúžiace na úpravu existujúcej cenovej ponuky."
api/roof/calculateNeededItems/{id}:
  post:
    summary: "POST api/roof/calculateNeededItems/{id}"
    operationId: "createRoof"
    parameters:
      - name: "id"
        in: "path"
        required: true
        schema:
          type: "string"
    responses:
      200:
```

Obr. C.5: 5

```
        description: "Api slúžiace na vytvorenie a priradenie položiek ku
streche."
  api/roof/createEmptyRoof:
    post:
      summary: "POST api/roof/createEmptyRoof"
      operationId: "createEmptyRoof"
      responses:
        200:
          description: "Api slúžiace na vytvorenie prázdnej strechy."
  api/roof/deleteRoof/{id}:
    delete:
      summary: "DELETE api/roof/deleteRoof/{id}"
      operationId: "deleteRoof"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na vymazanie strechy z databázy."
  api/roof/getAllRoofs:
    get:
      summary: "GET api/roof/getAllRoofs"
      operationId: "getAllRoofs"
      responses:
        200:
          description: "Api slúžiace na získanie všetkých striech z
databázy."
  api/roof/getRoofById/{id}:
    get:
      summary: "GET api/roof/getRoofById/{id}"
      operationId: "getRoofById"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na získanie strechy podľa jej id."
  api/roof/updateRoof/{id}:
    put:
      summary: "PUT api/roof/updateRoof/{id}"
      operationId: "updateRoof"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na úpravu strechy."
  api/store/createStore:
```

Obr. C.6: 6

```
    post:
      summary: "POST api/store/createStore"
      operationId: "createEmptyStore"
      responses:
        200:
          description: "Api slúžiace na vytvorenie prázdneho skladu."
  api/store/deleteStore/{id}:
    delete:
      summary: "DELETE api/store/deleteStore/{id}"
      operationId: "deleteStore"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na vymazanie skladu z databázy."
  api/store/getAllStores:
    get:
      summary: "GET api/store/getAllStores"
      operationId: "getAllStores"
      responses:
        200:
          description: "Api slúžiace na získanie všetkých skladov z
databázy."
  api/store/getStoreById/{id}:
    get:
      summary: "GET api/store/getStoreById/{id}"
      operationId: "getStoreById"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na získanie skladu z databázy na základe
id."
  api/store/updateStore/{id}:
    put:
      summary: "PUT api/store/updateStore/{id}"
      operationId: "updateStore"
      parameters:
        - name: "id"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na úpravu skladu."
  authority/createAuthority:
    post:
      summary: "POST authority/createAuthority"
```

Obr. C.7: 7

```
    operationId: "createAuthority"
    responses:
      200:
        description: "Api slúžiace na vytvorenie autority."
authority/deleteAuthority/{id}:
  delete:
    summary: "DELETE authority/deleteAuthority/{id}"
    operationId: "deleteAuthority"
    parameters:
      - name: "id"
        in: "path"
        required: true
        schema:
          type: "string"
    responses:
      200:
        description: "Api slúžiace na vymazanie autority"
authority/getAllAuthorities:
  get:
    summary: "GET authority/getAllAuthorities"
    operationId: "getAllAuthorities"
    responses:
      200:
        description: "Api slúžiace na získanie všetkých autorít"
authority/getAuthorityById/{id}:
  get:
    summary: "GET authority/getAuthorityById/{id}"
    operationId: "getAuthorityById"
    parameters:
      - name: "id"
        in: "path"
        required: true
        schema:
          type: "string"
    responses:
      200:
        description: "Api slúžiace na získanie autority na základe jej id."
authority/updateAuthority/{id}:
  put:
    summary: "PUT authority/updateAuthority/{id}"
    operationId: "updateAuthority"
    parameters:
      - name: "id"
        in: "path"
        required: true
        schema:
          type: "string"
    responses:
      200:
        description: "Api slúžiace na úpravu autority."
users/createUser:
  post:
    summary: "POST users/createUser"
    operationId: "createUser"
    responses:
      200:
        description: "Api slúžiace na vytvorenie nového používateľa."
```

Obr. C.8: 8

```
  users/deleteUser/{userName}:
    delete:
      summary: "DELETE users/deleteUser/{userName}"
      operationId: "deleteUser"
      parameters:
        - name: "userName"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na vymazanie používateľa z databázy."
  users/getAllUsers:
    get:
      summary: "GET users/getAllUsers"
      operationId: "getAllUsers"
      responses:
        200:
          description: "Api slúžiace na získanie všetkých používateľov."
  users/getUserByUserName/{username}:
    get:
      summary: "GET users/getUserByUserName/{username}"
      operationId: "getUserByUsername"
      parameters:
        - name: "username"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na získanie užívateľa na základe jeho
používateľského mena."
  users/updateUser/{userName}:
    put:
      summary: "PUT users/updateUser/{userName}"
      operationId: "updateUser"
      parameters:
        - name: "userName"
          in: "path"
          required: true
          schema:
            type: "string"
      responses:
        200:
          description: "Api slúžiace na úpravu užívateľa v databáze."
```

Obr. C.9: 9