Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Webová aplikace pro řízení provozu vertikální pěstírny na základě systému objednávek

Vojtěch Martin Kupka

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Vojtěch Martin Kupka

Informatika

Název práce

Webová aplikace pro řízení provozu vertikální pěstírny na základě systému objednávek

Název anglicky

Web application for vertical indoor growing room flow management based on ordering system

---

Cíle práce

Diplomová práce je tematicky zaměřena na vývoj webové aplikace implementující systém pro řízení vertikální pěstírny na základě objednávek od zákazníků.

Cílem práce je analýza požadavků pěstírny a následný návrh a modelování optimálního systému pro řízení chodu pěstírny. Navržené řešení pak bude realizováno ve formě webové aplikace. Aplikace bude představovat komplexní systém, který bude fungovat jako e-shop a zároveň bude pěstírně sloužit jako nástroj k řízení produkce.

Metodika

V teoretické části bude vypracován přehled problematiky single-page webových aplikací se zaměřením na tvorbu webových aplikací realizovaných ve frameworku Angular za použití jazyka TypeScript. Prostřednictvím modelování s využitím jazyka UML bude navržena komplexní webová aplikace včetně databáze, klíčové procesy v rámci aplikace budou vymodelovány dle standardů BPMN. Teoretická část se bude dále zabývat problematikou výhonků neboli tzv. microgreens v potravinářském průmyslu a využitím vertikálních pěstíren k jejich produkci.

V praktické části bude realizován vývoj a následné testování webové aplikace implementující navržené funkcionality pro provoz pěstírny. V rámci diskuse pak bude zhodnocena funkčnost a použitelnost vytvořené aplikace, její nedostatky a možný další vývoj celého systému. V závěru budou shrnuty výsledky celé práce.

Doporučený rozsah práce

60 – 80 stran

Klíčová slova

angular, typescript, javascript, microgreens, production management, UML, BPMN

Doporučené zdroje informací

FAIN, Yakov a Anton MOISEEV. Angular development with TypeScript. Second edition. Shelter Island: Manning, [2019]. ISBN 978-1-61729-534-8

FOWLER, Martin. Destilované UML. 1. vyd. Praha: Grada, 2009. ISBN 9788024720623

KHAN, Mohammad Ayoub, et al. (ed.). Smart Computing: Proceedings of the 1st International Conference on Smart Machine Intelligence and Real-Time Computing (SmartCom 2020), 26-27 June 2020, Pauri, Garhwal, Uttarakhand, India. CRC Press, 2021.

SAKS, Elar. JavaScript Frameworks: Angular vs React vs Vue. 2019.

STIEHL, Volker a SpringerLink (online služba). Process-Driven Applications with BPMN [online].1st 2014;2014;. Cham: Springer International Publishing, 2014. ISBN 9783319072180

Xiao, Zhenlei & Lester, Gene & Luo, Yaguang & Wang, Qin. (2012). Assessment of Vitamin and Carotenoid Concentrations of Emerging Food Products: Edible Microgreens. Journal of agricultural and food chemistry. 60. 7644-51. 10.1021/jf300459b.

XU, Wenqing. Benchmark Comparison of JavaScript Frameworks React, Vue, Angular and Svelte. 2021.

Předběžný termín obhajoby

2022/23 LS – PEF

Vedoucí práce

Ing. Václav Lohr, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 14. 7. 2022

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 28. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 25. 03. 2023

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Webová aplikace pro řízení provozu vertikální pěstírny na základě systému objednávek" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2023 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu práce Ing. Václavu Lohrovi, Ph.D. za odborné vedení a cenné připomínky při vypracovávání práce. Také bych rád poděkoval své rodině a přátelům za podporu a motivaci při studiu.

Webová aplikace pro řízení provozu vertikální pěstírny na základě systému objednávek

Abstrakt

Tato práce se věnuje návrhu a vývoji webové aplikace implementující systém pro řízení vertikální pěstírny na základě objednávek od zákazníků. V teoretické části práce je rozebírána problematika webových aplikací se zaměřením na framework Angular. Je zde popsána architektura a struktura Angular aplikací, problematika komponentů, datových vazeb v rámci komponentu a sdílení dat mezi jednotlivými komponenty. Teoretická část práce se dále věnuje serverovému prostředí Node.js a Express spolu s technologiemi pro realizaci architektury RestAPI. Vlastní práce se zabývá analýzou požadavků reálné pěstírny, na jejichž základě byl pomocí standardizovaných jazyků UML a BPMN navržen systém pro řízení chodu pěstírny. V rámci návrhu byla modelována struktura systému a klíčové procesy, které bude systém vykonávat. Navržený systém byl implementován formou webové aplikace za použití uvedených technologií. Vytvořená webová aplikace umožňuje zákazníkům vytvářet objednávky, na jejichž základě je řízen chod pěstírny. Provoz pěstírny je řízen prostřednictvím generování úkolů, které instruují obsluhu pěstírny. Cílem používání vytvořeného systému je optimalizace výrobního procesu a realizace štíhlé výroby v pěstírně.

Klíčová slova: angular, typescript, javascript, microgreens, řízení produkce, UML, BPMN

Web application for vertical indoor growing room flow management based on ordering system

Abstract

The aim of this graduation thesis is design and development of web application implementing a system for vertical growing room flow management based on ordering system. Theoretical part of the thesis is rooted in the issue of web applications with focus on the Angular framework along with the architecture and structure of Angular applications, problem of components, data bindings and data sharing between individual components. Theoretical part is also devoted to the server environment of Node.js and Express together with key technologies for realization of the RestAPI architecture. Main objective of this thesis is analysis of realistic growing room and design of flow management system based on the growing room requirements. As part of the application design, the structure of the system and the processes that the system would perform were modelled using the standardized languages UML and BPMN. Designed system was implemented in the form of a web application using the mentioned technologies. Created web application allows customers to create orders, based on which the growing room production flow is managed. Growing room flow is managed via generated tasks which instruct the operators of the growing room. The use of the created system could help to optimize production process and to allow implementation of lean production in the growing room.

Keywords: angular, typescript, javascript, microgreens, production management, UML, BPMN

Obsah

1 Úvod

Vznik celosvětové sítě www počátkem 90. let 20. století způsobil intenzivní přesun široké škály lidských činností do online prostředí [1]. Vývoj výpočetního modelu spolu s vytvořením programovacího jazyka JavaScript v roce 1995 umožnili vznik dynamických a interaktivních webových stránek, označovaných jako webové aplikace. Na přelomu milénia umožnil vývoj technologií webovým aplikacím komunikovat s webovými servery v reálném čase a aktualizovat části stránek bez nutnosti načíst celou stránku znova. [2, s. 7] To otevřelo cestu vývoji sofistikovanějších webových aplikací, které byly schopny spouštět kód v prohlížeči klienta, a tím poskytnout uživatelům mnoho funkcí, které byly dříve dostupné pouze v desktopových aplikacích. Webové aplikace přinášejí velké množství výhod, jako je nezávislost aplikací na platformě a typu zařízení. K webovým aplikacím uživatelé přistupují prostřednictvím internetového prohlížeče bez nutnosti předchozí instalace aplikace.

Rozvoj webových technologií a aplikací zásadně ovlivnil způsob života uživatelů internetu. Lidé dnes používají internet mimo jiné jako platformu pro komunikaci, vzdělávání, vyhledávání a sdílení informací, zábavu nebo pro výkon své profese. Většina uživatelů dnes přistupuje k webovému obsahu pomocí mobilních zařízení [3], která se stala každodenní součástí jejich životů. [4, s. 71]

Webové aplikace se staly nezbytnou součástí mnoha odvětví průmyslu. V oblasti výrobního průmyslu začaly být webové aplikace využívány jako nástroj k řízení a plánování výroby v reálném čase. Díky nim mohou podniky řídit výrobní procesy a plánovat produkci tak, aby docílily maximalizace výkonu a minimalizace ztrát. Takové aplikace jsou zároveň klíčové pro rozvoj průmyslu 4.0. [5]

Tato práce se zabývá návrhem a vývojem webové aplikace implementující systém pro řízení vertikální pěstírny na základě objednávek od zákazníků. Systém byl navržen a modelován pomocí standardů UML a BPMN na základě požadavků reálné pěstírny. Navržený systém byl dále implementován formou webové aplikace vytvořené za použití frameworku Angular, serverového prostředí Node.js Express a relační databáze MySQL.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je tematicky zaměřena na vývoj webové aplikace implementující systém pro řízení vertikální pěstírny na základě objednávek od zákazníků. Cílem práce je analýza požadavků pěstírny a následný návrh a modelování optimálního systému pro řízení chodu pěstírny. Navržené řešení pak bude realizováno ve formě webové aplikace. Aplikace bude představovat komplexní systém, který bude fungovat jako e-shop a zároveň bude pěstírně sloužit jako nástroj k řízení produkce.

2.2 Metodika

V teoretické části práce bude vypracován přehled problematiky single-page webových aplikací se zaměřením na tvorbu webových aplikací realizovaných ve frameworku Angular za použití jazyka TypeScript. Prostřednictvím modelování s využitím jazyka UML bude navržena komplexní webová aplikace včetně databáze, klíčové procesy v rámci aplikace budou vymodelovány dle standardů BPMN. Teoretická část se bude dále zabývat problematikou výhonků neboli tzv. microgreens v potravinářském průmyslu a využitím vertikálních pěstíren k jejich produkci.

V praktické části práce bude realizován vývoj a následné testování webové aplikace implementující navržené funkcionality pro provoz pěstírny. V rámci diskuse pak bude zhodnocena funkčnost a použitelnost vytvořené aplikace, její nedostatky a možný další vývoj celého systému. V závěru budou shrnuty výsledky celé práce.

3 Teoretická východiska

V teoretické části práce byla formou rešerše zpracována témata týkající se oblasti webových aplikací se zaměřením na framework Angular, Node.js a Express. V rámci rešerše byla zpracována problematika Angular aplikací, jejich architektura, struktura a některé aspekty jejich vývoje. Teoretická část práce se dále zabývala serverovým prostředím Node.js Express a dalšími technologiemi pro realizaci architektury RestAPI. V rámci teoretické části práce byla také zpracována problematika výhonků (microgreens) a jejich produkce v konkrétní pěstírně.

3.1 Framework Angular

Pojem framework, nebo také aplikační rámec, označuje obecně v informatice softwarovou strukturu, která zjednodušuje vývoj jiného softwaru. Framework může vývojáři nabízet návrhové vzory, knihovny, různé podpůrné programy a další výhody usnadňující vývoj aplikací. Angular je open-source framework pro tvorbu single-page webových aplikací založených na programovacím jazyce TypeScript. Pro svou komplexnost je Angular také označován jako platforma. Angular je spolu s technologiemi React a Vue řazen mezi nejpoužívanější front-end frameworky pro tvorbu webových aplikací. [6, s. 68-72], [7]

3.1.1 Historie frameworku

Předchůdcem frameworku Angular je framework AngularJS, který vytvořili Miško Hevery a Adam Abrons v roce 2009. AngularJS rozšiřuje HTML o nové možnosti, díky kterým je možné kombinovat deklarativní programování pro tvorbu uživatelského rozhraní s imperativním programováním pro tvorbu aplikační logiky. [8, s. 15-34] Angular vzniknul kompletním přepsáním frameworku AngularJS, první verze byla vytvořena v roce 2016 a byla pojmenována Angular 2, dnes známá jako Angular. Framework Angular je vyvíjen a udržován společností Google ve spolupráci s dalšími vývojáři a korporacemi.

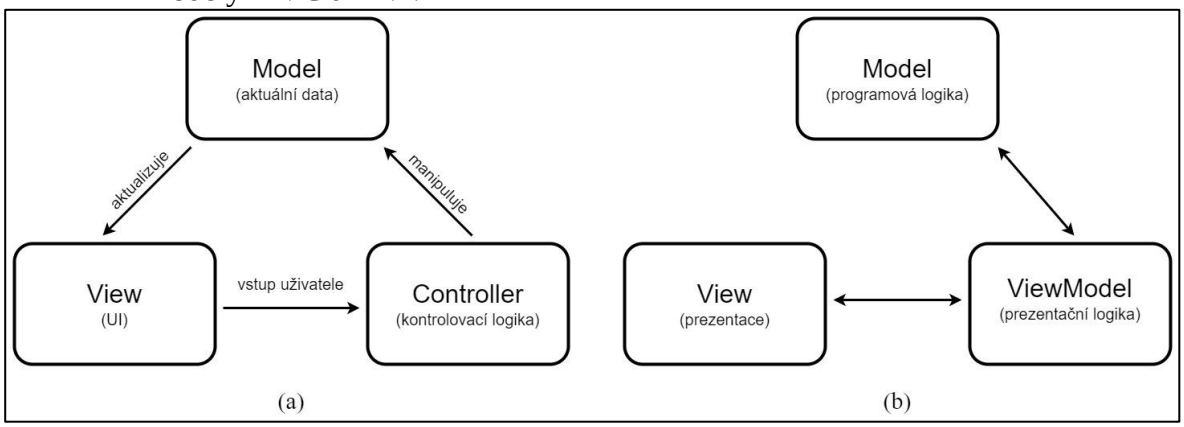
Angular CLI (command-line interface) je nástroj používaný pro zakládání, vývoj, skládání a údržbu Angular aplikací z prostředí příkazového řádku. Nainstalovat ho lze pomocí *npm* manažeru. Funguje jako generátor kódu, který značně usnadňuje tvorbu nových projektů, komponent nebo servisů. *Angular CLI* automaticky instaluje Angular framework a všechny potřebné závislosti, lze pomocí něj generovat balíky aplikace určené jak

k produkčním účelům, tak pro vývoj. Pomocí *Angular CLI* lze rovněž aktualizovat použité knihovny, ale také celý projekt do poslední verze Angularu. [9]

3.1.2 MVC a MVVM

Model-View-Controller (MVC) je návrhový vzor pro designování uživatelských rozhraní aplikací. Podle této šablony je aplikace rozdělena do tří částí, kterými jsou model, uživatelské rozhraní (View) a kontrolér (Controller). Díky tomuto rozdělení je umožněn vývoj aplikační logiky nezávisle na uživatelském rozhraní. Model uchovává aktuální data, zatím co uživatelské rozhraní funguje jako prezentační vrstva, která zobrazuje data uživateli. Kontrolér spojuje model s uživatelským rozhraním. Uživatelské rozhraní zobrazuje aktuální data z modelu, ale změny dat v uživatelském rozhraní se nepropagují zpět do modelu přímou cestou. Místo toho jsou informace o změnách v uživatelském rozhraní předávány kontroleru, který následně aktualizuje data v modelu. Změna dat v modelu se projeví aktualizací uživatelského rozhraní. Vzor MVC byl přijat jako vhodná architektura pro tvorbu single-page webových aplikací, mimo jiné také frameworkem Angular. [10, s. 13] Schéma MVC je zachyceno na obrázku 1 v části (a).

Obrázek 1 Modely MVC a MVVM



Zdroj: vlastní zpracování

Framework Angular využívá modifikaci klasického MVC, známou pod zkratkou MVVM (Model-View-ViewModel). Vazbu mezi uživatelským rozhraním (View) a modelem realizuje třída ViewModel. Třída ViewModel uchovává informace o stavu aplikace a je svázaná s uživatelským rozhraním. Uživatelské rozhraní se dotazuje ViewModelu, a podle něj zobrazuje uživateli ovládací prvky. ViewModel poskytuje uživatelskému rozhraní data v upravené struktuře, která při změně dat v uživatelském rozhraní vyvolává události propagující informace o změnách zpět do ViewModelu.

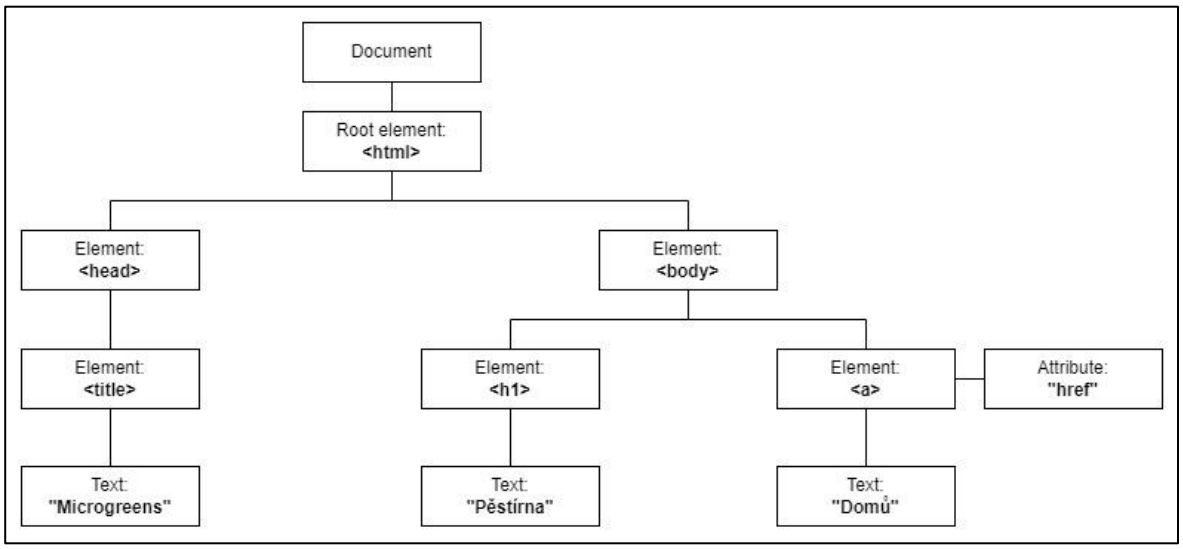
[11, s. 36-47] Model a uživatelské rozhraní spolu komunikují pouze prostřednictvím ViewModelu. Schéma MVVM je pro ilustraci zachycena na obrázku 1 v části (b).

3.1.3 DOM a Bootstrap

Document Object Model (DOM) je interface umožňující scriptům dynamicky přistupovat a měnit obsah, strukturu a styl dokumentů. Dle W3C standardu je DOM definován jako nezávislý jak na platformě, tak na programovacím jazyku. Pro HTML dokumenty je definován HTML DOM. [12] Jakmile dojde k načtení webové stránky, prohlížeč vygeneruje její DOM, který pak uchovává v paměti. DOM je konstruován jako strom objektů, kdy je každá větev zakončena uzlem (node) obsahujícím objekty. Strom obsahuje objekty HTML elementů a rovněž jejich vlastnosti, metody a události. Metody DOM umožňují programátorský přístup ke stromu a poskytují tak možnost získávat, měnit, přidávat nebo odstraňovat HTML elementy. [13] Ukázka struktury DOM je zachycena na obrázku 2.

Ovlivňování HTML elementů napřímo pomocí manipulace s DOM je časově náročné a má negativní dopad na výkonnost aplikace. Aby bylo dosaženo uživatelsky příjemného UX (user experience), je zapotřebí dosáhnout obnovovací frekvence 60 fps, což vyžaduje intenzivní manipulaci s DOM. [14] Proto bylo nutné náročné operace s DOM zjednodušit. Z tohoto důvodu využívá Angular Ivy inkrementální DOM (IDOM). IDOM je založen na principu, kdy je každý komponent kompilován do sad instrukcí. Tyto sady instrukcí pak vytvářejí stromy DOM a aktualizují je pouze na konkrétních místech, kde došlo ke změně dat. Prakticky je v paměti počítače vytvořen virtuální DOM (VDOM) na základě kterého se generuje reálný DOM. VDOM je aktualizován s každou změnou dat a je následně porovnáván s reálným DOM. Podle výsledků porovnání se pak aktualizují jen ty části DOM, kde došlo ke změně dat. [15, s. 153-156] Tento přístup mimo jiné výrazně zlepšuje použitelnost Angular aplikací na mobilních zařízeních. [16]

Obrázek 2 DOM



Zdroj: Převzato a upraveno [13]

Bootstrap je framework obsahující sadu nástrojů kaskádových stylů pro tvorbu UI webových aplikací. Usnadňuje a zrychluje vývoj aplikací díky šablonám založených na HTML a CSS, které jsou optimalizovány pro mobilní zařízení, tablety i stolní počítače. Bootstrap je založený na mobile-first přístupu a podporují jej všechny moderní prohlížeče. Z toho důvodu je používán také v rámci Angular aplikací. [17]

3.1.4 JavaScript, TypeScript a kompilace

Angular aplikace mohou být vyvíjeny v jazyce JavaScript nebo TypeScript. Framework Angular je z většiny napsaný v jazyce TypeScript a tento jazyk je při vývoji Angular aplikací preferován. Programovací jazyk TypeScript byl vydán v roce 2012 společností Microsoft a jeho hlavním vývojářem byl Anders Hejlsberg. Webové prohlížeče pracují pouze s jazykem JavaScript a nerozumějí proto některým specifickým prvkům jazyka TypeScript, jako jsou například strukturální směrnice **ngIf* a **ngFor*. TypeScript musí být tudíž do jazyka JavaScript transpilován (mezi vývojáři se častěji používá pojem kompilace). To znamená, že kód v jazyce TypeScript musí být přeložen do zdrojového kódu v jazyce JavaScript. Použití jazyka TypeScript zjednodušuje vývoj Angular aplikací a přináší řadu výhod a nových funkcionalit. [18, s. 3-4]

Kompilaci kódu z jazyka TypeScript zajišťuje Angular kompilátor (ngc). Kompilace může probíhat v prohlížeči, kdy je ngc přibalený jako součást aplikace v balíku *vendor.bundle.js*. Ke kompilaci dochází postupně při načítání příslušných částí aplikace. Tento typ kompilace je označován jako JIT (just-in-time) kompilace. Mezi načítáním balíků

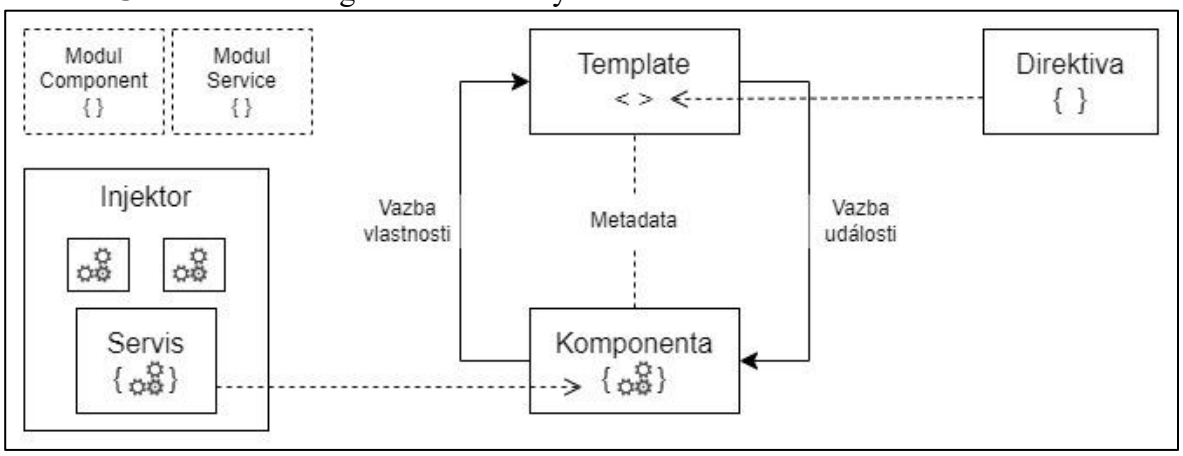
(bundles) aplikace a renderováním UI tak vzniká prodleva potřebná pro kompilaci kódu. Dochází rovněž k zvětšení velikosti aplikace, jelikož součástí aplikace musí být samotný ngc. [18, s. 13-15]

Alternativou je kompilace AOT (ahead-of-time), kdy dochází ke kompilaci kódu ještě před vytvořením balíků aplikace. Do prohlížeče se tudíž dostává již zkompilovaný kód, který prohlížeč rovnou zobrazí. Angular kompilátor ngc tudíž nemusí být součástí aplikace. V rámci frameworku Angular je doporučeno primárně využívat AOT kompilaci, která je nastavena jako defaultní. Jsou ale případy, kdy může být výhodnější použití JIT kompilace, například pokud je zkompilovaný kód výrazně větší než před kompilací. [19] O celý proces kompilace a renderování se stará technologie Angular Ivy. Jedná se o řetězec procesů, který mimo jiné zrychluje AOT kompilaci a vytváří inkrementální DOM. [20]

3.2 Architektura Angular aplikace

Architektura Angular aplikace je vystavěna na několika základních konceptech. Základními stavebními prvky frameworku Angular jsou komponenty, které jsou uspořádány do modulů. Moduly shromažďují související komponenty do funkčních celků a definují aplikaci. Komponenty definují svá zobrazení (Views), což jsou sady prvků obrazovky, mezi kterými Angular vybírá a zobrazuje je uživateli podle aplikační logiky, která je obsažena ve třídě komponentu. K realizaci funkcionalit, které nesouvisejí s konkrétním View, používají komponenty služby, které jsou do komponentů vkládány jako injekce závislostí. Moduly, komponenty a služby jsou třídy, které od sebe Angular odlišuje pomocí dekoračních funkcí. Dekorační funkce poskytují metadata modifikující a popisující příslušnou třídu. [21] Na obrázku 3 je vyobrazeno schéma popisující základní komponenty Angular aplikace a jejich vzájemnou komunikaci.

Obrázek 3 Schéma Angular architektury



Zdroj: Převzato a upraveno [21]

3.2.1 Architektura založená na komponentech

"We're not designing pages, we're designing systems of components. " - Stephen Hay

Pro člověka může být přirozené představit si webovou aplikaci z hlediska struktury jako tištěnou knihu skládající se z jednotlivých stran, mezi kterými můžeme listovat. Tato představa ale bohužel neodpovídá struktuře moderních webových aplikací. Na moderní webovou aplikaci je potřeba nahlížet jako na systém komponent, které jsou samy o sobě nezávislé, ale mohou spolu interagovat a skládat se do větších funkčních celků. [22, s. 3-8]

3.2.2 Konceptuální pohled na komponenty

Komponenty jsou samostatné softwarové objekty, které mezi sebou interagují. Základními vlastnostmi komponentů jsou zapouzdření (encapsulation) a skládání (composability). Zapouzdření komponentů má velký přínos pro údržbu systému a pro přístup ke komponentům. Z hlediska OOP lze zapouzdření chápat jako sdružení logiky a dat do jednoho kontejneru, se kterým je pak možné manipulovat jako s jedním celkem. Podle stejného principu je možné sdružovat také jednotlivé komponenty. To napomáhá mimo jiné k lepší organizaci zdrojového kódu. [23, s. 1-16]

Skládání je možné chápat jako možnost opakovaně používat komponenty. Princip skládání spočívá v tom, že místo rozšiřování jednoho komponentu je nový větší komponent vytvořen složením z více menších komponentů. Chování nového komponentu pak bude sjednocením chování dílčích komponentů, ze kterých byl nový komponent složen. [23, s. 1-16]

Příroda je tím nejlepším učitelem. Množství úspěchů na poli technologického vývoje je založeno na pozorování přírody a jejím následném napodobování. Všechny organismy v přírodě se musejí neustále vyvíjet a přizpůsobovat se nejrůznějším změnám. Gion Kunz ve své knize Mastering Angular Components přirovnává aplikaci k organismu, který se musí rovněž vyvíjet a vyrovnávat se změnami, které nevyhnutelně přináší čas. Autor knihy se zamýšlí nad tím, že právě zapouzdření a skládání jsou klíčovými koncepty, díky kterým se aplikace dokážou vyrovnávat se změnami a vyvíjet se bez velkých ztrát na účinnosti. [22, s. 9]

Jednotlivé komponenty si lze pro ilustraci představit jako dílky stavebnice LEGO. Jeden dílek stavebnice může být opakovaně používán v jiných projektech. Vždy se spojí dohromady s ostatními dílky, čímž vznikne něco nového. Navíc, pokud bychom se rozhodli vyměnit například červený dílek za modrý, můžeme to udělat bez obav, že by do stavebnice nezapadnul. Stejně tak se dají komponenty jedné aplikace použít v dalších aplikacích. Komponent lze rovněž nahradit jiným bez toho, aby byly ovlivněny ostatní komponenty. Pokud vybereme konkrétní dílek stavebnice, je snadné ho prozkoumat a zkontrolovat. Možnost zaměřit se na konkrétní komponent zjednodušuje jeho vývoj a následné testování. Všichni stavitelé stavějí z jedné společné krabice s dílky. Každý může dílky ve společné krabici používat, upravovat nebo přidávat dílky nové. To přináší další výhody, například stavitelé postupem času vědí, jaké dílky společná krabice obsahuje a jaký dílek se jim nejlépe hodí. [24]

3.2.3 Komponenty a moduly

Z hlediska frameworku Angular se pojmem komponent (component) rozumí třída, ovládající část obrazovky nazvanou View. K tomu, aby byla třída rozpoznána jako komponent, musí být označena dekorační funkcí *@Component()*. [18, s. 20-23] Komponent se skládá z třídy obsahující aplikační logiku, HTML templatu a listu CSS stylů. Třída komponentu definuje interakci mezi HTML templatem a DOM, zatímco CSS styly upravují vzhled UI. [21] Komponenty jsou základními stavebními kameny Angular aplikací a lze je chápat jako autonomní celky, ze kterých je aplikace poskládána. [25, s. 68] Angular aplikaci si tudíž lze představit jako strom komponentů. Rovněž musí existovat kořenový komponent, který funguje jako kontejner pro všechny ostatní komponenty. Neexistuje striktní pravidlo, z kolika komponentů by měla být aplikace tvořena, nicméně vývojáři se pokoušejí aplikace rozdělovat na co největší množství komponentů, tudíž tvořit

jednotlivé komponenty co nejmenší. [10, s. 20-22] Deklarace komponentu je zachycena v ukázce kódu 1.

Ukázka kódu 1 Komponent

```
// importy
import { Component } from '@angular/core';

// dekorační funkce
@Component({
  selector: 'app-component',
  templateUrl: './ component.component.html',
  styleUrls: ['./ component.component.css']
})

// třída komponentu
export class Childcomponent2Component {
  constructor() { }
}
```

Zdroj: vlastní zpracování

Angular aplikace jsou díky komponentům modulární, stejně jako celý framework Angular. Angular používá vlastní modulární systém nazvaný NgModuly. NgModuly shromažďují související kód a utvářejí z něj jeden celek. NgModul je prakticky kontejner pro komponenty, servisy a další soubory obsahující kód, které spolu logicky souvisejí. [26] Díky modulům je možné přehledně organizovat aplikaci a rozdělovat ji na menší části. Pomocí modulů je také možné přidávat do aplikace funkcionality z externích knihoven. Z hlediska syntaxe je NgModul třída označená dekorační funkcí *@NgModule()*. Dekorační funkce obsahuje objekt s metadaty, popisující třídu modulu. Obsahuje deklarace všech součástí modulu, jeho vstupy, výstupy a poskytovatele služeb. Výstupy jsou podmnožinou deklarací součástí modulu, které by měly být viditelné a použitelné v rámci templatu komponentů ostatních modulů. [27] Každá Angular aplikace obsahuje kořenový modul, a může obsahovat libovolné množství dalších modulů. Aplikace se spouští bootstrapováním kořenového NgModulu. Všechny moduly aplikace se nemusejí načítat najednou, ale díky routeru mohou být načítány postupně, až když jsou potřeba (lazy-loading). To výrazně zlepšuje výkonnost aplikace. Deklarace modulu je zachycena v ukázce kódu 2.

Ukázka kódu 2 Modul

```
// importy
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { MessageComponent } from './message/message.component';

@NgModule({
  // deklarace komponent
  declarations: [
    AppComponent,
    MessageComponent
  ],
  imports: [
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Zdroj: vlastní zpracování

3.2.4 Template a View

Template je zdrojový kód, definující, jak se má vykreslovat View komponentu. Vypadá jako standardní HTML, ale je doplněn o speciální prvky, které propojují template s aplikační logikou a stavem DOM. [28] Template je spojen s třídou komponentu pomocí dekorační funkce *@Component()*. Třída komponentu a přidružený template definují View, které lze chápat jako nejmenší seskupení prvků obrazovky, které může být vytvořeno nebo skryto najednou jako jeden celek. Vlastnosti jednotlivých elementů v rámci View se mohou měnit v závislosti na interakci uživatele, ale jejich struktura nikoliv. [29] Jednotlivá View jsou typicky součástí hierarchie, která umožňuje manipulovat s částmi obrazovky jako s jedním celkem, například je zobrazovat nebo skrývat. Při kombinování View různých komponent se není potřeba obávat konfliktu jejich stylů, protože Angular umí zapouzdřit styly komponentů k elementu tak, aby neovlivňovaly zbytek aplikace. Zapouzdření je možné nastavit v dekorační funkci komponentu. [30] Zapouzdření View realizuje Shadow DOM. Jedná se o technologii webového prohlížeče, která funguje jako API poskytující zapouzdření při implementaci komponentů. [31] Ukázce kódu 3 ukazuje příklad HTML templatu.

Ukázka kódu 3 HTML Template

```
<h2>View</h2>
<p>Zadané jméno: <b>{{ name }}</b></p>
<input [(ngModel)]="name">
<div *ngIf="true"><p>Tento obsah se zobrazí ve View.</p></div>
<div *ngIf="false"><p>Tento obsah se nezobrazí ve View.</p></div>
<div *ngIf="false"><app-childcomponent1></app-childcomponent1>
    <p>Tato komponenta se nezobrazí ve View.</p>
</div>
```

Zdroj: vlastní zpracování

3.2.5 Služby, direktivy a svislice

Služby (services) jsou třídy označeny dekorační funkcí *@Injectable()*. Není k nim přiřazeno žádné UI a obsahují kód s logikou na UI nezávislou, která může být používána napříč celou aplikací. [32] Služby se používají převážně k získávání dat a manipulaci s nimi. Jsou pro tento účel vhodnější než klasické komponenty a lze díky nim lépe rozdělovat kód aplikace, což přidává na její přehlednosti. [18, s. 23] Komponent může být závislý na jedné nebo více službách. Jednotlivé služby jsou do komponent vkládány jako injekce závislostí. [33] Deklarace služby je zachycena v ukázce kódu 4.

Ukázka kódu 4 Služba

```
// importy
import { Injectable } from '@angular/core';

// dekorační funkce
@Injectable({
  providedIn: 'root'
})

// třída služby
export class TestService {
  constructor() { }
}
```

Zdroj: vlastní zpracování

Direktivy (directives) rozšiřují standardní HTML o nové prvky a funkcionality. Direktiva je třída, která je schopna modifikovat strukturu DOM a datového modelu komponentu. Direktivu označuje dekorační funkce *@Directive()*. [34] Mezi direktivy jsou řazeny také komponenty. Komponenta je speciálním rozšířením direktivy, kde je ke třídě direktivy přiřazen template. Dále jsou rozlišovány direktivy atributů a strukturální direktivy, které se odlišují od komponentů právě tím, že nemají přiřazen vlastní template. Direktivy atributů upravují chování a vzhled prvků na stránce, zatímco strukturální direktivy upravují strukturu DOM. Angular nabízí škálu integrovaných direktiv, ale je také možné vytvářet nové direktivy implementující vlastní funkcionalitu. [35]

Svislice (pipes) transformují řetězce, měny, datumy a ostatní data do požadovaných formátů pro zobrazování na obrazovce. Jedná se o jednoduché funkce používané v rámci templatu, které na vstupu přijímají hodnotu a na výstupu vracejí její příslušnou transformaci. Angular nabízí množství běžně používaných vestavěných svislic pro implementaci často realizovaných transformací. [36] K zapouzdření transformací, které nejsou poskytovány vestavěnými svislicemi, lze vytvořit vlastní svislice. Ty jsou realizovány jako třídy označené dekorační funkcí *@Pipe()*. Speciální zabudovanou svislicí je *AsyncPipe*, která přijímá na vstupu proměnnou typu *Observable* a automaticky nastaví její subskripci. Využívá se například při komunikaci se serverovou částí aplikace pomocí HTTP. [37]

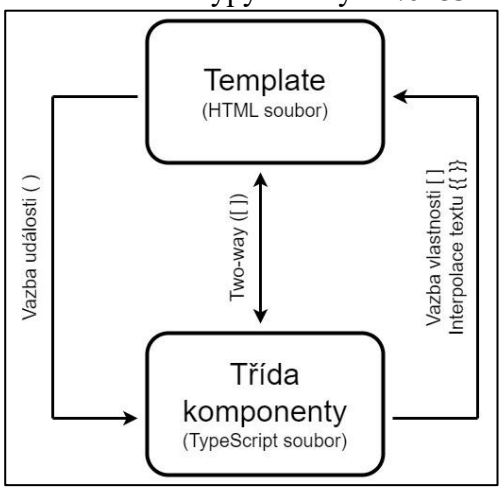
3.3 Datové vazby

Datové vazby (data bindings) vytvářejí živé spojení mezi View a třídou komponentu, čímž udržují UI aktualizované se stavem aplikace. Datová vazba je mechanismus, pomocí kterého lze propagovat změny proměnných ve třídě komponentu do View a obráceně, čímž se udržuje model a View v synchronizovaném stavu. Jinými slovy lze pomocí tohoto spojení automaticky aktualizovat View a informovat model o událostech a akcích uživatele. [38]

O udržování View a modelu v synchronizovaném stavu se stará algoritmus pro detekci změn. Jedná se o proces, pomocí kterého Angular kontroluje, zda došlo ke změně ve stavu aplikace a zda je potřeba aktualizovat nějaký DOM element. Detekce změn může být spuštěna manuálně nebo skrze asynchronní událost. Přesto že jde o vysoce optimalizovaný proces, může způsobovat zpomalení aplikace, pokud je prováděn příliš často. [39]

Podle směru toku dat rozlišuje Angular dva druhy datových vazeb. Prvním typem vazby jsou vazby jednosměrné (unidirectional). Jednosměrná vazba může přenášet data směrem ze třídy komponentu do View. V tomto směru jsou realizovány vazby atributů, tříd, stylů, interpolační vazby a vazby vlastností. V opačném směru, tedy z View do třídy komponentu, jsou realizovány vazby událostí. Druhým typem vazby je vazba obousměrná (two-way), která je realizována z View do třídy komponentu a následně zpátky do View. [40] Schéma typů datových vazeb je zachyceno na obrázku 4.

Obrázek 4 Typy datových vazeb

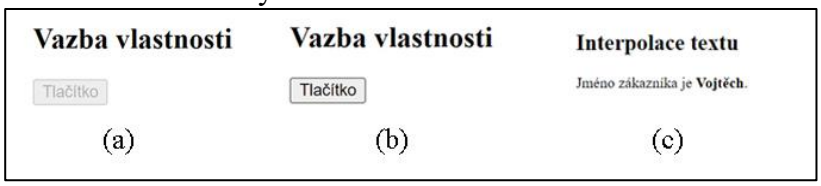


Zdroj: vlastní zpracování

3.3.1 Vazby vlastností a interpolace textu

Pomocí vazeb vlastností (property binding) je možné v Angularu nastavit hodnoty vlastností HTML elementů nebo direktiv v DOM. Hodnota se přesouvá jednosměrně z třídy komponentu do cílového elementu. K vytvoření vazby vlastnosti elementu je zapotřebí cílový element uzavřít do hranatých závorek *[ ]*. Vazba vlastnosti cílí na název vlastnosti v rámci DOM, nikoliv přímo na název atributu. [41] Vhodným příkladem použití může být zpřístupnění tlačítka. Tlačítko reprezentuje HTML button element, který má v rámci DOM vlastnost *disabled*. Pokud je tato vlastnost platná, na tlačítko nelze kliknout. Pro zpřístupnění tlačítka je potřeba změnit hodnotu proměnné *uzamcen* nacházející se ve třídě komponentu, s kterou je vlastnost *disabled* svázána. Změna se bude propagovat do View a tlačítko se uživateli zpřístupní. Vazbu vlastnosti zachycují ukázky kódu 5 a 6, na obrázku 5 lze pozorovat zobrazení View v prohlížeči. Část (a) obrázku 5 ukazuje stav, kdy je proměnná *uzamcen* nastavena na hodnotu *true*, část (b) obrázku 5 ukazuje situaci, kdy má proměnná *uzamcen* hodnotu *false*.

Obrázek 5 Vazby vlastností



Zdroj: vlastní zpracování

Ukázka kódu 5 Vazba vlastnosti: component.ts

```
uzamcen = true;
```

Zdroj: vlastní zpracování

Ukázka kódu 6 Vazba vlastnosti: component.html

```
<h2>Vazba vlastnosti</h2>
<button type="button" [disabled]="uzamcen">Tlačítko</button>
```

Zdroj: vlastní zpracování

Interpolace textu (text interpolation) je forma datové vazby vlastnosti, která umožňuje hodnoty proměnných ve třídě komponentu vykreslovat do templatu jako text. K tomu je potřeba výraz v templatu odkazující na proměnnou uzavřít do dvojitých složených závorek *{{ }}*. [42] V následujícím příkladu je ve třídě komponentu proměnná *zakaznik* obsahující jméno zákazníka, které je pomocí interpolace textu vykresleno do UI. Interpolaci textu zachycují ukázky kódu 7 a 8, na obrázku 5 v části (c) lze pozorovat zobrazení interpolace v prohlížeči.

Ukázka kódu 7 Interpolace textu (component.ts)

```
zakaznik = "Vojtěch"
```

Zdroj: vlastní zpracování

Ukázka kódu 8 Interpolace textu (component.html)

```
<h2>Interpolace textu</h2>
<p>Jméno zákazníka je <b>{{ zakaznik }}</b>.</p>
```

Zdroj: vlastní zpracování

3.3.2 Vazby atributů, tříd a stylů

Vazba atributu (attribute binding) cílí přímo na atributy HTML elementů a umožňuje nastavovat jejich hodnoty. Syntax se podobá vazbě vlastností, kdy se místo vlastnosti elementu nachází v hranatých závorkách název atributu s prefixem *attr*, který je s názvem atributu spojen tečkou. Vazby atributů umožňují dynamické spravování více CSS tříd nebo stylů. [43] Pomocí vazby atributu je možné například přidávat na stránku ARIA atributy, které se používají pro lepší orientaci hendikepovaných uživatelů na webu. Interpolaci textu zachycují ukázky kódu 9 a 10, na obrázku 6 v části (a) lze pozorovat zobrazení v prohlížeči. Část (b) obrázku 6 ukazuje HTML zdrojový kód zobrazený ve vývojářské konzoli webového prohlížeče.

Ukázka kódu 9 Vazba atributu (component.ts)

```
login = "Přihlásit se"
```

Zdroj: vlastní zpracování

Ukázka kódu 10 Vazba atributu (component.html)

```
<h2>Vazba ARIA atributu</h2>
<button type="button" [attr.aria-label]="login">
{{login}} s Aria atributem</button>
```

Zdroj: vlastní zpracování

Obrázek 6 Vazba atributu

Zdroj: vlastní zpracování

Vazby tříd a stylů (class and style binding) fungují podobně jako vazby atributů. Rozdílem však je, že cílí na třídy a styly. Pomocí těchto vazeb lze přidávat nebo odebírat názvy tříd z atributu *class* HTML elementů, na které reaguje Bootstrap, a dynamicky tak měnit jejich styly. [44] Následující kód přidává název třídy *approved* do atributu *class* HTML button elementu a mění barvu jeho pozadí na zelenou. Vazbu třídy a stylu zachycují ukázky kódu 11 a 12, na obrázku 7 v části (a) lze pozorovat zobrazení v prohlížeči. Část (b) obrázku 7 ukazuje HTML kód zobrazený ve vývojářské konzoli.

Ukázka kódu 11 Vazba třídy a stylu (component.ts)

```
zkontrolovan = "true";
color = "lightgreen";
```

Zdroj: vlastní zpracování

Ukázka kódu 12 Vazba třídy a stylu (component.html)

```
<h2>Vazba třídy a stylu</h2>
<button type="button" [class.approved]="zkontrolovan" [style.background-color]="color">
Zelené tlačítko</button>
```

Zdroj: vlastní zpracování

Obrázek 7 Vazba třídy a stylu

Zdroj: vlastní zpracování

3.3.3 Vazby událostí

Vazby událostí (event binding) umožňují sledovat akce uživatele a adekvátně na ně reagovat. Takovou akcí může být například pohyb a kliknutí myši nebo stisknutí klávesy. K realizaci vazby události využívá Angular syntax, kdy na levé straně rovnítka vystupuje cílová událost ohraničená jednoduchou závorkou *( )* a na pravé straně je výraz templatu v uvozovkách. Výraz templatu je obecně metoda nebo vlastnost používaná v HTML templatu k reagování na akce uživatele. [45], [46]

Vazbu události lze rovněž demonstrovat na příkladu s tlačítkem. Pokud uživatel klikne na tlačítko, aplikace zobrazí upozornění. Kliknutí na tlačítko spouští událost *(click)*, která volá metodu *zobrazUpozorneni()* ve třídě komponentu. Vazbu vlastnosti zachycují ukázky kódu 13 a 14, na obrázku 8 v části (a) lze pozorovat zobrazení tlačítka v prohlížeči. Část (b) obrázku 8 zachycuje akci vyvolanou stisknutím tlačítka.

Ukázka kódu 13 Vazba události (component.ts)

```
zobrazOznameni() {
    alert('Tlačítko bylo stisknuto.')
  }
```

Zdroj: vlastní zpracování

Ukázka kódu 14 Vazba události (component.html)

```
<h2>Vazba události</h2>
<button type="button" (click)="zobrazOznameni()">Zobrazit oznámení</button>
```

Zdroj: vlastní zpracování

Obrázek 8 Vazba události

Zdroj: vlastní zpracování

3.3.4 Obousměrné vazby

Obousměrná vazba (two-way binding) kombinuje vazbu události s vazbou vlastnosti. K přenosu dat tak dochází zároveň z View (UI) do modelu (třída komponentu) a z modelu zpět do View. Data ve View a modelu jsou tak neustále udržována v synchronizovaném stavu. [47]

V rámci komponentu lze realizovat obousměrnou vazbu pomocí direktivy *NgModel*. Tato direktiva je instancí třídy FormControl modulu FormsModule, který je zapotřebí

do aplikace importovat a deklarovat jej v hlavním modulu aplikace. K vytvoření direktivy se používá syntax, kdy na levé straně rovnítka vystupuje název direktivy uzavřený v závorkách *[( )]* a na pravé straně vlastnost třídy komponentu v uvozovkách. V následujícím příkladu je uživatel vyzván k zadání svého jména pomocí HTML input elementu. Změny elementu se propagují do třídy komponentu a pomocí interpolace textu se uživateli zobrazují v UI. Obousměrnou vazbu zachycují ukázky kódu 15 a 16, na obrázku 9 v části (a) lze pozorovat zobrazení v prohlížeči. Část (b) obrázku 9 vystihuje stav po zadání jména.

Ukázka kódu 15 Obousměrná vazba (component.ts)

```
name = "Zadejte své jméno";
```

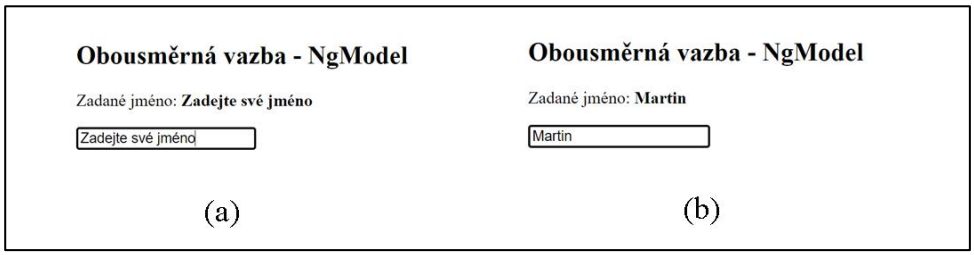
Zdroj: vlastní zpracování

Ukázka kódu 16 Obousměrná vazba (component.html)

```
<h2>Obousměrná vazba - NgModel</h2>
<h3>Zadané jméno: {{ name }}</h3>
<input [(ngModel)]="name">
```

Zdroj: vlastní zpracování

Obrázek 9 Obousměrná vazba



Zdroj: vlastní zpracování

3.4 Sdílení dat

V rámci aplikace je zapotřebí sdílet data mezi jednotlivými komponenty. Pro sdílení dat mezi komponenty je klíčové, zdali a jaký mají zainteresované komponenty vzájemný vztah. V Angular aplikacích je možné vložit určitý komponent do templatu jiného komponentu pomocí selektoru. Vložený komponent pak označujeme jako dceřiný (child) a komponent, kterému náleží template, je označován jako mateřský (parent). Sdílení dat mezi mateřským a dceřiným komponentem je možné pomocí dekoračních funkcí *@Input()* a *@Output()*. S využitím dekorační funkce *@ViewChild()* je pak možné injektovat dceřiný komponent do komponentu mateřského, který tak získá přístup k funkcím a vlastnostem dceřiného komponentu. Pokud komponenty nemají vzájemný vztah, je zapotřebí pro sdílení dat využít sdílenou službu.

3.4.1 Reaktivní programování

Reaktivní programování je paradigma orientované kolem datových toků a šíření změn. Datové toky by mělo být možné vyjádřit jako statické (pole) nebo dynamické (události). Rovněž by mělo být možné popsat odvozené závislosti v rámci modelu a usnadnit tak automatické šíření toku změněných dat. Nechť je proměnná a, která vznikla jako součet proměnných b + c. Pokud se změní hodnota libovolného ze sčítanců, automaticky se díky propagaci změny aktualizuje hodnota součtu, tedy závislé proměnné a. Toto chování je známé například z jinak funkcionálního programovacího jazyka Excel. Při použití MVVM architektury se mohou díky reaktivnímu programování změny v základním modelu automaticky propagovat do View, ale také šířit do dalších komponentů. [48, s. 7-9]

V rámci frameworku Angular je reaktivní programování realizováno díky knihovně RxJS (Reactive Extensions for JavaScript). Knihovna RxJS používá typ proměnné nazývaný *observable* (pozorovatelný). Tento typ proměnné se dá takzvaně pozorovat a díky tomu je možné implementovat asynchronní programovací logiku nebo logiku založenou na zpětné vazbě. RxJS rovněž nabízí řadu služeb pro práci s proměnnými typu *observable*. [49] Angular využívá ngRX, což je reaktivně orientovaný set modulů využívající RxJS k implementaci logiky reaktivního programování.

Reaktivní programování je známé paradigma, na kterém je postavena architektura založená na událostech EDA (Event-driven architecture). EDA je přístup k softwarové architektuře založený na vytváření, detekci, zpracování a reakci na události, přičemž událost je možné chápat jako změnu stavu. Pomocí událostí lze přenášet informace mezi komponentami nebo službami, kde vždy vystupuje producent (emitter) a konzumet (consumer) události. Událostmi řízená architektura může doplňovat servisně orientovanou architekturu, kde služby mohou být spouštěny na základě událostí. [50]

3.4.2 Posílání dat do dceřiného komponentu

Prostřednictvím dekorační funkce *@Input()* mohou být hodnoty proměnných dceřiném komponentu nastavovány podle vlastností komponentu mateřského. Pro ilustraci je možné například zobrazit jméno zadané uživatelem do mateřského komponentu v komponentu dceřiném. K realizaci takové vazby je potřeba do dceřiného komponentu nejdříve importovat knihovnu *Input* a následně označit cílovou vlastnost příslušnou dekorační funkcí *@Input()*. Dceřiný komponent se vkládá do templatu mateřského komponentu pomocí svého selektoru. S využitím vazby vlastnosti je navázána vlastnost

jmeno dceřiného komponentu na vlastnost *zadaneJmeno* mateřského komponentu. Pro získání jména zadaného uživatelem v mateřském komponentu je použita obousměrná vazba vlastnosti *zadaneJmeno*. [51] Ukázky kódu 17 a 18 znázorňují dceřiný komponent, do něhož jsou posílána data z mateřského komponentu. Mateřský komponent je zachycen v ukázkách kódu 19 a 20. Výsledek v prohlížeči zachycuje obrázek 10.

Ukázka kódu 17 @Input(): child.component.ts

```
@Input() jmeno = '';
```

Zdroj: vlastní zpracování

Ukázka kódu 18 @Input(): child.component.html

```
<div class="child">
    <h2>Dceřiná komponenta</h2>
    <p>Zadané jméno je: <b>{{ jmeno }}</b></p>
</div>
```

Zdroj: vlastní zpracování

Ukázka kódu 19 @Input(): parent.component.ts

```
zadaneJmeno = "";
```

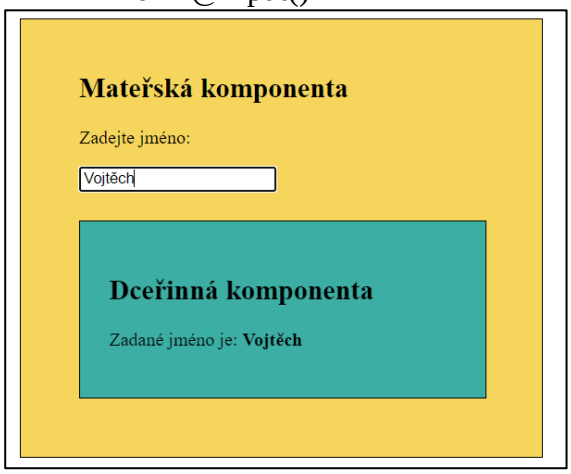
Zdroj: vlastní zpracování

Ukázka kódu 20 @Input(): parent.component.html

```
<div class="parent">
    <h2>Mateřská komponenta</h2>
    <input [(ngModel)]="zadaneJmeno">
    <app-childcomponent1 [jmeno]="zadaneJmeno"></app-childcomponent1>
</div>
```

Zdroj: vlastní zpracování

Obrázek 10 @Input()



Zdroj: vlastní zpracování

3.4.3 Posílání dat do mateřského komponentu

Pro posílání dat z dceřiného do mateřského komponentu je využívána dekorační funkce *@Output()*. Dekorační funkce *@Output()* označuje vlastnost dceřiného komponentu, skrze kterou mohou data cestovat do mateřského komponentu a zároveň spouští událost,

která informuje mateřský komponent o změně označené vlastnosti dceřiného komponentu. [51]

Případ užití je možné ilustrovat situací, kdy uživatel zadá své jméno do View dceřiného komponentu a odešle jej do komponentu mateřského, který jej zobrazí ve svém View. K tomu je zapotřebí do dceřiného komponentu nejdříve importovat knihovnu *Output* a *EventEmitter* a následně označit cílovou vlastnost *zmenaJmena* dekorační funkcí *@Output()*. Označená vlastnost je typu *EventEmitter*, jedná se tedy o událost. Ve třídě dceřiného komponentu se dále implementuje metoda *aktualizovatJmeno()*, která použije *@Output()* k vyvolání události obsahující hodnotu jména, kterou odešle do mateřského komponentu. Uživatel zadá jméno pomocí input elementu a stisknutím tlačítka se vyvolá metoda *aktualizovatJmeno()*. Ukázky kódu 21 a 22 znázorňují dceřiný komponent, ze kterého jsou odesílána data do mateřského komponentu znázorněného v ukázkách kódu 23 a 24. Výsledek v prohlížeči zachycuje obrázek 11.

Ukázka kódu 21 @Output: child.component.ts

```
@Output() zmenaJmena = new EventEmitter<string>();

aktualizovatJmeno(value: string) {
    this.zmenaJmena.emit(value);
  }
```

Zdroj: vlastní zpracování

Ukázka kódu 22 @Output: child.component.html

```
<div class="child">
    <h2>Dceřiná komponenta</h2>
    <label for="item-input">Nové jméno:</label>
    <input type="text" id="item-input" #noveJmenoInput>
    <button type="button"
(click)="aktualizovatJmeno(noveJmenoInput.value)">Odeslat</button>
</div>
```

Zdroj: vlastní zpracování

Ve třídě mateřského komponentu je implementována metoda *zmenitJmeno()*, která zpracuje vyvolanou událost obsahující zadané jméno a uloží jej do proměnné *noveJmeno*, kterou zobrazí uživateli pomocí interpolace textu. Do templatu mateřského komponentu je vložen template dceřiného komponentu pomocí selektoru. V templatu mateřského komponentu je pomocí vazby události spojena událost dceřiného komponentu *zmenaJmena* s metodou mateřského komponentu *zmenitJmeno()*, která událost zpracovává.

Ukázka kódu 23 @Output: parent.component.ts

```
noveJmeno = "";

zmenitJmeno(newItem: string) {
    this.noveJmeno = newItem;
  }
```

Zdroj: vlastní zpracování

Ukázka kódu 24 @Output: parent.component.html

```
<div class="parent">
    <h2>Mateřská komponenta</h2>
    <p>Nové jméno je: <b>{{ noveJmeno }}</b></p>
    <app-childcomponent1 (zmenaJmena)="zmenitJmeno($event)"></app-childcomponent1>
</div>
```

Zdroj: vlastní zpracování

Obrázek 11 @Output

Zdroj: vlastní zpracování

Využitím dekorační funkce *@ViewChild()* je možné injektovat dceřiný komponent do komponentu mateřského. Díky tomu může mateřský komponent přistupovat k vlastnostem dceřiného komponentu, a navíc může volat jeho veřejné metody. [52] Injektované proměnné nejsou dostupné okamžitě, ale až jakmile se inicializuje View. Dokončení inicializace View lze zachytit pomocí háku životního cyklu komponenty *AfterViewInit()*, v rámci kterého je možné přistupovat k injektovaným proměnným. [53] Ukázky kódu 25 a 26 znázorňují dceřiný komponent, který je injektován do mateřského komponentu znázorněného v ukázkách kódu 27 a 28. Výsledek v prohlížeči zachycuje obrázek 12.

Ukázka kódu 25 @ViewChild(): parent.component.ts

```
@ViewChild(Childcomponent1Component) child;
message:string = "";

ngAfterViewInit(): void {
    this.message = this.child.message;
  }
```

Zdroj: vlastní zpracování

Ukázka kódu 26 @ViewChild(): parent.component.html

```
<div class="parent">
    <h2>Mateřská komponenta</h2>

    <p>Jméno je: <b>{{ message }}</b></p>
    <app-childcomponent1></app-childcomponent1>
</div>
```

Zdroj: vlastní zpracování

Ukázka kódu 27 @ViewChild(): child.component.ts

```
message:string = "Vojtěch";
```

Zdroj: vlastní zpracování

Ukázka kódu 28 @ViewChild(): child.component.html

```
<div class="child">
    <h2>Dceřiná komponenta</h2>
</div>
```

Zdroj: vlastní zpracování

Obrázek 12 @ViewChild()

Zdroj: vlastní zpracování

3.4.4 Sdílení dat mezi komponenty

V situacích, kdy je zapotřebí sdílet data mezi komponenty, které mezi sebou nemají přímý vztah, lze pro sdílení dat využít sdílenou službu. K udržení dat v synchronizovaném stavu je možné využít *BehaviorSubject* z knihovny *RxJS*. Výhodou je, že *RxJS BehaviorSubject* neustále udržuje poslední hodnotu, kterou emituje ihned po subskripci proměnné. Rovněž zajišťuje, aby byla odeslána aktuální hodnota do všech komponent. [54] Pro ilustraci lze vytvořit komponent, který odešle zprávu do dalších komponentů, které

zprávu zobrazí. Nejdříve je však zapotřebí vytvořit sdílenou službu. Tato služba obsahuje *BehaviorSubject zdrojZprava*. Dále obsahuje proměnnou *aktualniZprava* zpracovávající datový tok zpráv, která je typu *observable* a může tedy být odebírána ostatními komponenty. Nakonec obsahuje funkci *novaZprava()*, která nastavuje novou hodnotu proměnné *zdrojZprava*. Deklaraci služby zachycuje ukázce kódu 29.

Ukázka kódu 29 Sdílená služba: service.ts

```
private zdrojZprava = new BehaviorSubject<string>("Původní zpráva");
aktualniZprava = this.zdrojZprava.asObservable();

novaZprava(zprava: string) {
    this.zdrojZprava.next(zprava);
  }
```

Zdroj: vlastní zpracování

Do komponentu odesílajícího zprávu se importuje služba, která se následně injektuje do konstruktoru třídy komponentu. Komponent musí dále obsahovat funkci *odeslatZpravu()*, která nastaví novou hodnotu zprávy zavoláním funkce *novaZprava()* vytvořené služby. Funkce se zavolá po stisknutí tlačítka. Komponent odesílající zprávu je vystižen v ukázkách kódu 30 a 31.

Ukázka kódu 30 Sdílená služba: unrelated.component.ts

```
constructor (private data: MessageService) { }

odeslatZpravu() {
    this.data.novaZprava("Nová zpráva!");
  }
```

Zdroj: vlastní zpracování

Ukázka kódu 31 Sdílená služba: unrelated.component.html

```
<div class="komponenta">
    <h2>Komponenta odesílající zprávu</h2>
    <button type="button" (click)="odeslatZpravu()">Odeslat zprávu</button>
</div>
```

Zdroj: vlastní zpracování

Do komponentů přijímajících zprávu, které mají v tomto případě mezi sebou přímý vztah, se rovněž importuje vytvořená služba, která se injektuje do konstruktoru komponentu. Pomocí háku životního cyklu *ngOnInit()* začne komponent při inicializaci odebírat proměnnou *aktualniZprava* vytvořené služby, která obsahuje hodnotu poslední odeslané zprávy. Získaná hodnota se uloží do proměnné *zprava*, jejíž hodnota se v UI zobrazuje pomocí interpolace textu. Komponent přijímající data je zobrazen v ukázkách kódu 32 a 33, výsledek v prohlížeči ukazuje obrázek 13.

Ukázka kódu 32 Sdílená služba: acceptor.component.ts

```
zprava!: string;
constructor (private data: MessageService) { }

ngOnInit() {
    this.data.aktualniZprava.subscribe(zprava => this.zprava = zprava)
  }
```

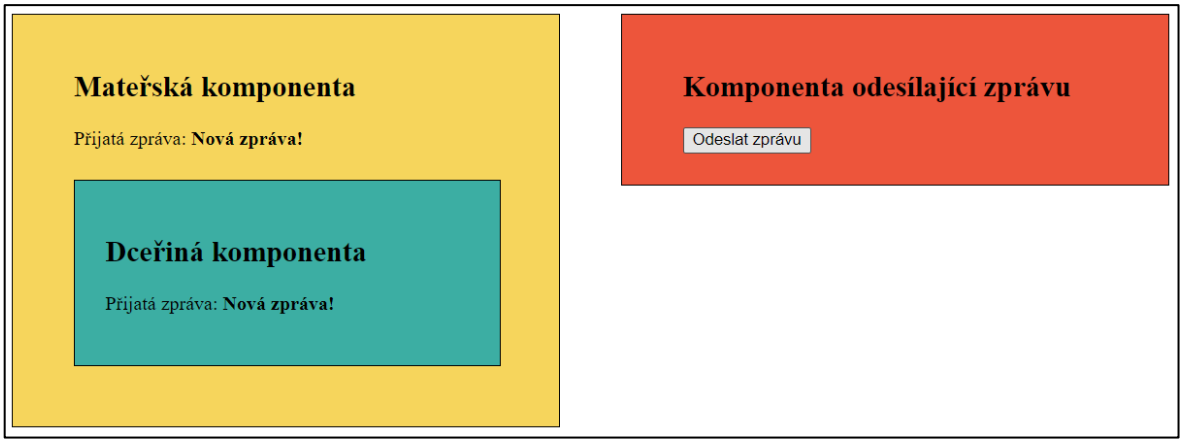
Zdroj: vlastní zpracování

Ukázka kódu 33 Sdílená služba: acceptor.component.html

```
<div class="child">
    <h2>Dceřinná komponenta</h2>
    <p>Přijatá zpráva: <b>{{ zprava }}</b></p>
</div>
```

Zdroj: vlastní zpracování

Obrázek 13 Sdílená služba



Zdroj: vlastní zpracování

3.5 Serverová část aplikace

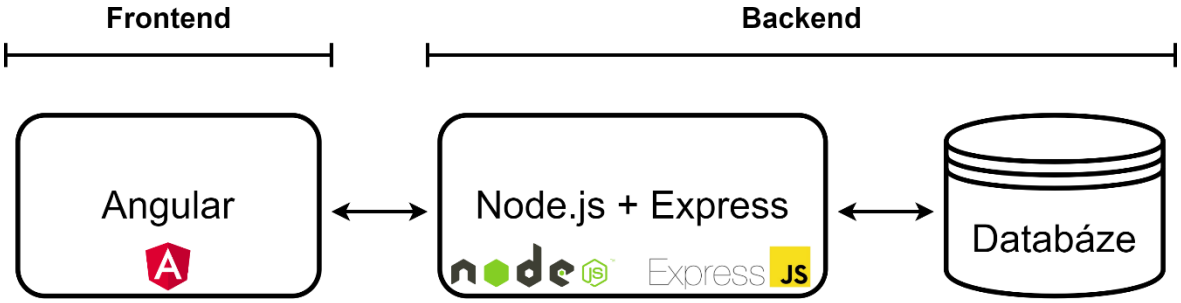
Serverová část aplikace je označována souhrnným názvem backend. Jedná se o část aplikace, kterou uživatel nevidí a nemá k ní přístup. Backend je obecně vrstva aplikace realizující přístup k datům. Je v kontrastu s frontendem, který lze definovat jako prezentační vrstvu. Backend generuje obsah webové stránky a odesílá jej do prohlížeče uživatele. Lze tedy říct, že vše, co se stane před zobrazením stránky v prohlížeči uživatele, realizuje backend.

3.5.1 Node.js a Express

Node.js je open-source multiplatformní prostředí pro běh programů umožňující vývojářům vytvářet server-side aplikace pomocí jazyka JavaScript. Node byl vytvořen jako nezávislý na prostředí prohlížeče, tudíž může běžet přímo v operačním systému počítače nebo serveru. [55] Jednou z jeho hlavních výhod je rychlost. Jádro Node je napsáno v C/C++ a JavaScript kompiluje do strojového kódu pomocí rychlého a výkonného kompilátoru V8.

Dalším důvodem je asynchronní architektura, díky které může paralelně zpracovávat více procesů. Node navíc obsahuje množství integrovaných knihoven a modulů, například knihovnu *RxJS* nebo modul HTTP, umožňující hostovat webový server. [56, s. 50-62] Node lze chápat mimo jiné jako nástroj umožňující používat JavaScript na straně serveru. To umožňuje vývojářům tvořit kompletní webovou aplikaci za použití jednoho programovacího jazyka. Schéma takové webové aplikace je pro ilustraci zachyceno na obrázku 14.

Obrázek 14 Schéma webové aplikace



Zdroj: vlastní zpracování

Express.js je minimalistický a flexibilní framework pro serverové prostředí Node.js nabízející širokou škálu funkcionalit pro vývoj webových a mobilních aplikací. Express také tvoří základ mnoha knihoven a frameworků pro Node.js. [56, s. 71-73] Pojmem minimalistický se rozumí fakt, že Express realizuje jen minimalistickou vrstvu, do které lze dle potřeby přidávat další knihovny. Framework Express je nedogmatický, což znamená, že vývojářům nenabízí ověřené postupy, ale nechává je nalézt si vlastní řešení. Express zjednodušuje práci s Node a výrazně zkracuje délku kódu. [57, s. 8-12]

3.5.2 Komunikace prostřednictvím HTTP

Angular aplikace komunikují se všemi typy webových serverů podporujících HTTP (Hypertext Transfer Protocol), nezávisle na platformě, na které server funguje. Tuto komunikaci umožňuje architektura RestAPI (Representational Státe Transfer Application Programming Interface). Jedná se o architekturu webových služeb, která umožňuje komunikaci mezi klientem a serverem pomocí HTTP protokolu. RestAPI využívá standardní *HTTP* metody a formát dat. [58, s. 97] Pro realizaci komunikace se serverem nabízí Angular službu *HttpClient*, která funguje jako client-side API umožňující provádět HTTP dotazy. Mezi hlavní funkce služby *HttpClient* se řadí schopnost dotazovat se na objekty deklarovaného typu, zachytávání dotazů a odpovědí, zjednodušené řešení chyb a nástroje

pro testování. [59] Pomocí služby *HttpClient* lze tedy odesílat různé typy HTTP požadavků, jako jsou *GET, POST, PUT, DELETE* a další. *HttpClient* poskytuje také možnost zpracovávat odpovědi v různých formátech, jako je například formát JSON (JavaScript Object Notation). JSON je formát pro přenos dat, který je snadno čitelný pro lidi i stroje.

Pro zabezpečený přenos informací mezi aplikacemi v rámci webového prostředí lze využít JSONWebToken (JWT). JWT se používá pro ověření identity uživatele webových aplikací. Když se uživatel přihlásí, server vytvoří JWT, který obsahuje informace o uživateli a jeho oprávnění. JWT se skládá ze tří částí: hlavičky (header), těla (payload) a podpisu (signature). [60] Hlavička obsahuje informace o typu tokenu a použitém algoritmu pro jeho podepsání. Tělo obsahuje uživatelská data nebo jiné informace, které se mají přenést, a podpis, vytvořený pomocí soukromého klíče, sloužící k ověření pravosti tokenu. Tento token je po vytvoření předáván v každém dalším požadavku, který uživatel na server posílá. Server vždy ověří pravost tokenu, a pokud je token platný, umožní uživateli provádět požadované operace. JWT má několik výhod oproti tradičním metodám ověřování identity, jako je například ukládání dat do souborů cookies. To umožňuje ověřit pravost tokenu bez nutnosti ukládat data na straně serveru.

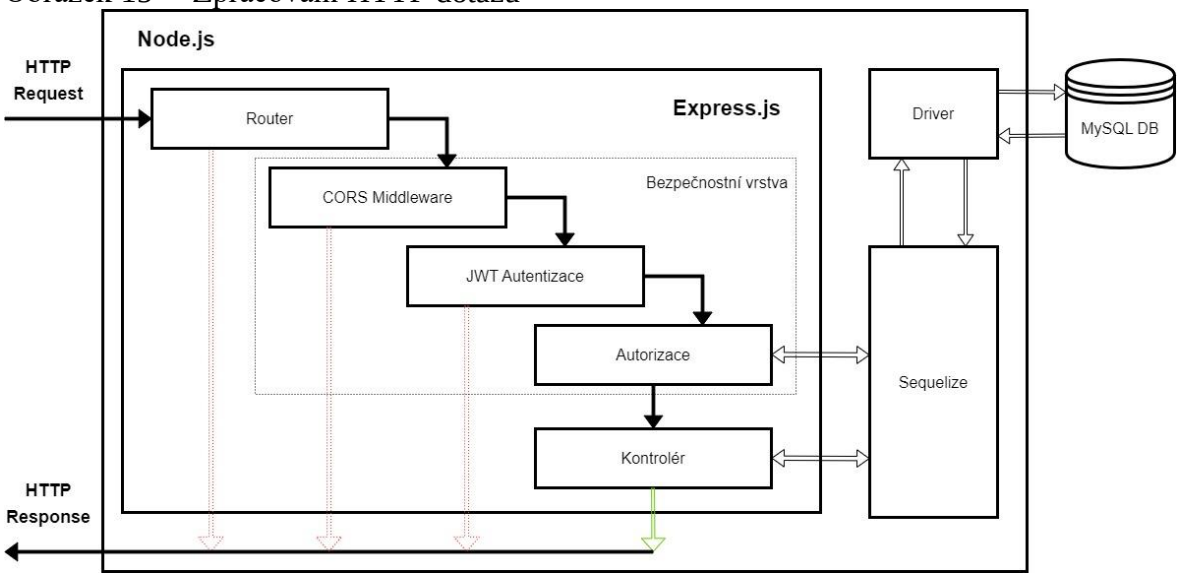
Ukládání dat o probíhající relaci do cookies umožňuje middleware modul *cookie-session*. Jedná se o technologii, díky které je možné po přihlášení uživatele vygenerovat unikátní JWT, který je následně uložen do cookies ve webovém prohlížeči klienta. Následné HTTP požadavky klienta pak obsahují vytvořený token v hlavičce požadavku, což umožňuje serverové části aplikace u každého požadavku ověřit autorizaci uživatele a na daný požadavek pak následně patřičně reagovat. [58, s. 90]

Aby HTTP požadavek obsahoval zmíněný JWT, je zapotřebí jej v klientské části aplikace modifikovat pomocí rozhraní *HttpInterceptor*. Jedná se o rozhraní umožňující aplikaci modifikovat HTTP požadavky a odpovědi, které procházejí přes službu *HttpClient*. Díky tomuto rozhraní je možně mj. přidávat autentizační JWT tokeny do hlaviček HTTP požadavků.

Průběh zpracování požadavků je vyobrazen na obrázku 15. Jedná se o schéma, na kterém lze pozorovat interakce komponent implementující zmíněné technologie v serverové části aplikace. Příchozí HTTP požadavek je nejdříve zpracován routerem. Router je součástí frameworku Node.js Express a jeho funkcí je mapování URL požadavků na konkrétní funkce, které mají být vykonány v serverové části aplikace. Po nalezení příslušné cesty prochází požadavek přes CORS (Cross-Origin Resource Sharing) middleware. CORS

je bezpečnostní mechanismus frameworku Express, který umožňuje definovat, jaké zdroje a metody jsou povoleny pro přístup na server. Následuje proces autentizace, kdy dojde k vyhodnocení JWT v hlavičce požadavku. Pokud je token platný, následuje autorizace uživatele, při které dojde k ověření oprávnění přiděleného uživateli. Tyto procesy společně vytvářejí bezpečnostní vrstvu serverové části aplikace. Pokud některý z ověřovacích procesů v rámci bezpečnostní vrstvy skončí chybou, zpracovávání HTTP požadavku je ukončeno a chybové hlášení je odesláno zpět do klientské části aplikace. Pokud požadavek úspěšně projde bezpečnostní vrstvou, provede se aplikační logika příslušné funkce obsažené v kontroléru. Výstup funkce kontroléru je následně odeslán do klientské části aplikace.

Obrázek 15 Zpracování HTTP dotazu



Zdroj: vlastní zpracování

Angular aplikace provádějí HTTP dotazy asynchronně. Díky tomu je UI responsivní a umožňuje uživateli pracovat s aplikací, zatímco se HTTP požadavky zpracovávají. Asynchronní dotazování pomocí HTTP lze realizovat mimo jiné pomocí proměnných typu *observable*. [18, s. 133-150]

3.5.3 Databáze

Databázi lze definovat jako uspořádanou soustavu dat. Ve světě počítačů se v současnosti často využívají relační databáze. Podle relačního modelu je databáze definována jako soustava tabulek, ve kterých jsou uložena data uspořádána do řádků (záznamů) a sloupců (polí). Hodnoty uložené ve sloupcích jsou pro některé tabulky společné a tvoří tak mezi nimi vztahy (relace). Pro práci s relačními databázemi vyvinula firma IBM

dotazovací jazyk SQL (Structured Qvery Language). Z hlediska informatiky je databáze soubor dat, který je zpracováván pomocí nějakého software. Software zpracovávající databázi je označován jako databázový systém. Databázový soubor obsahuje různé objekty, hlavními jsou tabulky, pohledy a diagramy. Pohledy jsou předdefinované příkazy jazyka SQL, které lze opakovaně spouštět. V diagramech jsou uloženy závislosti a vztahy tabulek v databázi. [61]

Framework Express umožňuje práci s různými typy databázových systémů. Aktuálním trendem je využívání NoSQL databází, například MongoDB. Jedná se o typ nerelační databáze, kde jsou data uložená v objektech typu JSON, místo v tabulkách. Pro účely této práce byla použita relační databáze spravovaná databázovým software MySQL. Přesto zde bylo v rámci aplikace k relační databázi přistupováno pomocí objektově orientovaných paradigmat. To je možné díky technologii Sequelize. Jedná se o open-source ORM (Object-Relational Mapping) umožňující vývojářům pracovat s databází pomocí jazyka JavaScript místo SQL dotazů. Sequelize ORM umožňuje programátorům definovat modely databáze v jazyce JavaScript, které se pak mapují na relační schéma a dotazy. [58] Sequelize dále poskytuje sadu funkcí pro vytváření, čtení, aktualizaci a mazání záznamů v databázi, řazení, filtrování a agregaci dat a také podporuje relační vztahy mezi tabulkami.

3.6 Microgreens

Microgreens jsou výhonky plodin určené ke konzumaci. Plodiny se sklízejí v ranné fázi po vyrašení prvních pravých lístků. Lze je také definovat jako sklizené výhonky listové zeleniny. Microgreens jsou oblíbené v gastronomickém průmyslu kvůli svému atraktivnímu vzhledu, jelikož disponují širokou paletou barev a chutí. Zlepšují chuťové vlastnosti a nutriční hodnoty jídel, jelikož obsahují množství antioxidantů, vitamínů, minerálů a dalších bioaktivních složek. Jsou tudíž považovány za "funkční potraviny", které podporují zdraví člověka a prevenci nemocí. [62] Microgreens ve fázi sklizně jsou zobrazeny na obrázku 16.

Obrázek 16 Microgreens

Zdroj: vlastní zpracování

3.6.1 Nutriční hodnoty microgreens

Microgreens obsahují minerály, vitamíny a další tělu prospěšné bioaktivní látky. Jako doplněk stravy tak mohou alternovat jiné syntetické suplementy. Co se týče koncentrace bioaktivních složek, pozorujeme u microgreens často násobně vyšší koncentrace tělu prospěšných látek než u dospělých rostlin. Například u výhonků červeného hlávkového zelí pozorujeme šestkrát vyšší koncentraci vitamínu C a šedesátkrát vyšší koncentraci vitamínu K než v dospělé rostlině. [63] Toto však nelze označit za pravidlo, jelikož například u výhonků špenátu setého pozorujeme vyšší koncentrace vitamínu A, ale nižší koncentrace betakarotenu než u dospělé rostliny. [64] Obsahy vitamínů a minerálů ve výhoncích různých rostlinných druhů se v literatuře zabývají například [65] nebo [62]. V tabulce 1 uvádíme přehled koncentrací vybraných vitamínů a minerálů pro vybrané druhy výhonků. Hodnoty koncentrací jsou uvedeny v mg / 30 g výhonků, což odpovídá běžné porci.

Tabulka 1 Koncentrace vitamínů a minerálů u vybraných výhonků

Rostlinný druh	Vitamíny			Minerály								
	C	E	K	Ca	Cu	Fe	Mg	Mn	P	Na	K	Zn
Slunečnice roční *Helianthus annuus*	26	14,7		27		420	15		21	48	100	75
Ředkev setá bílá *Raphanus sativus var. Longipinnatus*	29	5,8	54	20	23	186	20	81	29	19	70	120
Kedluben bílý *Brassica oleracea var. Gongylodes*	20	4,2	69	27	30	225	17	115	23	10	102	130
Červené hlávkové zelí *Brassica oleracea var. capitata F. rubra*	45	7,2	84	22	23	186	13	93	22	11	70	113
Cibule kuchyňská *Allium cepa*	9	4,5		10		297	6,9		9	12	81	40

Zdroj: [63] [62] [65] [64]

3.6.2 Pěstování microgreens

Tato kapitola se zabývá procesem pěstování microgreens v konkrétní pěstírně. Uvedené informace vycházejí z reálného provozu konkrétní pěstírny, nemusí být tudíž obecně platné. V rámci pěstebního procesu microgreens je rozlišováno celkem 5 fází: sorbování vody, klíčení, vršení, růst a sklizeň. Rostlinné druhy se od sebe odlišují jak délkou jednotlivých fází růstu, tak v některých případech také pořadím těchto fází. Jako pěstební médium je použit univerzální výsadbový substrát o výšce 1,5 – 2 cm. Pěstební plato se substrátem je zobrazeno na obrázku 17.

Obrázek 17 Pěstební plato se substrátem

Zdroj: vlastní zpracování

Semena některých rostlinných druhů vyžadují pro dosažení maximální klíčivosti nejdříve kontakt s vodou. Při absorpční fázi jsou semena ponořena do odstáté vody o pokojové teplotě. Absorpce vody způsobuje bobtnání semen, což napomáhá prolomení semenného obalu, čímž se výrazně zkrátí doba trvání následného klíčení. Absorpce vody rovněž pozitivně ovlivňuje efektivitu klíčení, tudíž zvětšuje poměr vyklíčených a nevyklíčených semen. Absorpční fázi obecně vyžadují rostlinné druhy, které se vyznačují

větší velikostí semen (například hrách setý nebo slunečnice roční). Délka sorbování vody je specifická pro každý rostlinný druh. Při nedosažení požadované doby sorbování dojde k nedostatečnému podpoření klíčivosti. Při překročení doby sorbování hrozí tzv. utopení semen. Taková semena pak mají problémy s klíčením a vytvářejí tak podmínky pro vznik plísní.

Na absorpci vody plynule navazuje fáze klíčení (BO). Pro většinu druhů se jedná o zahajovací fázi pěstebního procesu. Semena jsou při ní rovnoměrně rozprostřena po pěstebním médiu. Semena by měla být lehce zatlačena do substrátu, nikoliv však zasypána. Takto připravené pěstební pláty se semeny jsou umisťovány do zatemňovacích domů, ve kterých je udržována relativní vzdušná vlhkost v rozmezí 90–95 %. Fáze klíčení je klíčovým momentem v celém pěstebním procesu, jelikož semena při ní musejí nejen vyklíčit, ale také zakořenit. V této fázi si microgreens zachovávají zpravidla žlutou barvu z důvodu absence světla. Délka fáze klíčení se pohybuje od 3 do 7 dnů a u některých druhů je doplněna o fázi vršení. Výhonky ve fázi klíčení jsou zachyceny na obrázku 18.

Obrázek 18 Microgreens: fáze klíčení

Zdroj: vlastní zpracování

U některých rostlinných druhů je fáze klíčení doplněna fází vršení (ST). Tato fáze se obecně využívá u druhů s většími semeny, jejichž kořeny kvůli své velikosti obtížně nacházejí cestu mezi ostatními kořeny a dochází k neefektivnímu zakořenění. Proto se ve fázi vršení naskládají pěstební pláta na sebe a jsou zatížena závažím. Tím je docíleno

rovnoměrného zakořenění všech semen. Tato doplňující fáze trvá v rozmezí 1-2 dnů. Po dokončení fáze vršení může následovat opět fáze klíčení. Semena jsou sice již zakořeněna a vyklíčena, ale ponecháním klíčků bez přístupu světla se docílí jejich vertikálního protažení. To je u některých druhů nutné pro dosažení jejich požadované velikosti. Pokud jsou klíčky po skončení fáze vršení dostatečně vzrostlé, následuje fáze růstu. Výhonky ve fázi navršení zachycuje obrázek 19.

Obrázek 19 Microgreens: fáze vršení

Zdroj: vlastní zpracování

Při růstové fázi (LO) jsou výhonky vystaveny světlu a začíná probíhat fotosyntéza. Pro všechny druhy microgreens je v prvních dvou dnech této fáze typické jejich zabarvení do odstínů barev zelené, červené a fialové. Následuje pozvolný růst (0,5-1cm/den). Pravidelné rosení je nahrazeno zaléváním pomocí zavlažovacího pláta, které je vloženo pod pláto pěstební. Mezi pláty tak vzniká prostor pro vytvoření bohatého kořenového systému. Délka růstové fáze je obecně 3-7 dnů v závislosti na konkrétním rostlinném druhu. Při růstové fázi je klíčové zejména adekvátní dávkování vody. Při přemokření začne docházet k uhnívání kořenů, což má za následek úhyn výhonků. Výhonky ve fázi růstu jsou zachyceny na obrázku 20.

Obrázek 20 Microgreens: fáze růstu

Zdroj: vlastní zpracování

Poslední fází pěstebního procesu je fáze sklizně. Tato fáze nastává, jakmile microgreens dosáhnou požadované velikosti a jsou připraveny ke konzumaci. Zde je důležité provést sklizeň v optimální čas. Uplynutí lhůty pro sklizení u výhonků značí vznik pravých listů. Po sklizení lze střižené výhonky skladovat v lednici, nicméně během skladování ztrácejí microgreens na kvalitě. Tato relativně krátká doba skladování je brána jako jedna z nevýhod microgreens. Ideální je konzumovat microgreens čerstvě sklizené. Výhonky připravené ke sklizení jsou zachyceny na obrázku 21.

Obrázek 21 Microgreens: fáze sklizně

Zdroj: vlastní zpracování

V pěstírně se produkují truhlíky s danou kombinací microgreens, ale také truhlíky na míru. Kombinaci microgreens v každém truhlíku na míru sestavuje sám zákazník při objednávce. Truhlík na míru může obsahovat až 4 různé rostlinné druhy dle preference zákazníka. Jak už bylo zmíněno, jednotlivé rostlinné druhy se od sebe odlišují jak délkou jednotlivých fází růstu, tak v některých případech také pořadím těchto fází. Přechody mezi jednotlivými fázemi růstu realizuje obsluha pěstírny odpovídajícími úkony, jako je například navršení truhlíků s klíčícími semeny nebo vystavení vyklíčených výhonků světlu. Pro maximální efektivitu a výnos je potřeba tyto úkony provádět v přesně stanovený čas nebo se mu alespoň co nejvíce přiblížit. Jelikož různé rostlinné druhy vyžadují odlišný čas růstu, musí být semena do truhlíku seta postupně tak, aby v den předání truhlíku zákazníkovi byly všechny výhonky ideálně vzrostlé. Jednotlivé fáze růstu pěstovaných výhonků a jejich délku zjednodušeně ukazuje tabulka 2.

Tabulka 2 Fáze růstu vybraných výhonků

Rostlinný druh	Den růstu											
	1	2	3	4	5	6	7	8	9	10	11	12
Hrách setý * *Pisum sativum*	BO	BO	ST	ST	BO	LO	LO	LO	LO	LO		
Slunečnice roční * *Helianthus annuus*	BO	BO	ST	ST	BO	LO	LO	LO				
Ředkev setá bílá *Raphanus sativus var. Longipinnatus*	BO	BO	BO	ST	LO	LO						
Ředkev setá červená *Raphanus sativus*	BO	BO	ST	LO	LO							
Kedluben bílý *Brassica oleracea var. Gongylodes*	BO	BO	BO	ST	LO	LO						
Červené hlávkové zelí *Brassica oleracea var. capitata F. rubra*	BO	BO	BO	ST	LO	LO						
Laskavec trojbarevný *Amaranthus tricolor*	BO	BO	BO	BO	BO	BO	LO	LO	LO	LO	LO	LO
Cibule kuchyňská *Allium cepa*	BO	BO	BO	BO	BO	BO	BO	LO	LO	LO	LO	LO

Zdroj: vlastní zpracování

4 Vlastní práce

Praktická část práce se věnuje návrhu a vývoji webové aplikace implementující systém pro řízení vertikální pěstírny na základě objednávek od zákazníků. Proces vývoje začíná analýzou požadavků reálné pěstírny a následným definováním základních funkcionalit požadovaného systému. Praktická část pokračuje návrhem a modelováním optimálního systému a jeho implementací formou single-page webové aplikace.

4.1 Analýza požadavků pěstírny

Pěstírna produkuje truhlíky s výhonky. Každý pěstovaný truhlík je rozdělen do čtyř částí o stejné ploše, přičemž každá část může obsahovat odlišný druh výhonku. Složení truhlíku určuje zákazník při vytváření objednávky. Ukázka vypěstovaného truhlíku je znázorněna na obrázku 22. Navržená aplikace by měla zákazníkovi nabídnout intuitivní a vizuálně atraktivní UI pro sestavení truhlíku a vytvoření objednávky. Pro realizaci objednávky by se měl zákazník nejdříve registrovat anebo se přihlásit ke svému již existujícímu účtu. Vypěstovaný truhlík bude zákazníkovi doručen na jeho adresu, tudíž by měl mít zákazník možnost kdokoliv upravit své kontaktní údaje a aktualizovat adresu doručení.

Obrázek 22 Ukázka truhlíku

Zdroj: vlastní zpracování

Jak bylo uvedeno v kapitole 3.6.2 (Pěstování microgreens), jednotlivé druhy výhonků mají specifickou celkovou dobu růstu, růstové fáze a délky trvání jednotlivých růstových fází. K tomu, aby mohly výhonky rostoucí v některé ze čtyř částí truhlíku přejít

do navazující fáze růstu, je vyžadována aktivita ze strany obsluhy pěstírny. Takovou aktivitou může být například založení nového truhlíku, vysetí semen do určité části truhlíku nebo odkrytí určité části truhlíku, čímž se klíčky v dané části vystaví světlu.

Truhlík znázorněný na obrázku 22 obsahuje zleva následující čtyři rostlinné druhy: ředkev setá bíla, ředkev setá červená, kedluben bílý a hrách setý. Nejdéle rostoucím druhem obsaženým v tomto truhlíku je hrách setý rostoucí ve 4. části truhlíku, jehož celková doba růstu je 10 dnů. Odtud se odvíjí celková doba potřebná pro vypěstování celého truhlíku, která je rovna 10 dnům od vysetí semen hrachu setého. První den pěstebního procesu se tedy vysejí semena hrachu setého do 4. určené části truhlíku, čímž dojde k založení truhlíku a odstartování jeho pěstebního cyklu. Při této aktivitě musí obsluha pěstírny zaznamenat číslo reálného truhlíku, do kterého byla semena vyseta. Třetí den se klíčící semena hrachu setého ve 4. části truhlíku zatíží, čímž započne fáze vršení klíčků. Pátý den pěstebního procesu se do 1. části truhlíku zasejí semena ředkve bílé a do 3. části truhlíku semena kedlubny bílé. Zároveň se klíčky hrachu setého ve 4. části truhlíku přesunou do fáze sekundárního klíčení. Šestý den se do 2. části truhlíku zasejí semena ředkve červené a zároveň se klíčky hrachu setého ve 4. části truhlíku přesunou do fáze růstu na světle, při které začne probíhat fotosyntéza. Osmý den pěstebního procesu se klíčky v 1., 2. a 3. části truhlíku zatíží a přejdou tak do fáze vršení. Devátý den se všechny klíčky nacházející se ve fázi vršení odkryjí a vystaví se světlu. Desátý den je truhlík připraven pro předání zákazníkovi, jelikož výhonky ve všech čtyřech částech truhlíku jsou připraveny ke sklizni. Jednotlivé manipulační aktivity s výhonky v jednotlivých částech truhlíku v průběhu pěstebního procesu vycházejí z tabulky 2 vyobrazené v kapitole 3.6.2 (Pěstování microgreens).

Navržený systém pro řízení pěstírny by měl na základě analýzy příchozí objednávky naplánovat aktivity potřebné k vypěstování truhlíku a převést je na úkoly, které budou v příhodné dny zobrazovány obsluze pěstírny. Takový systém by měl zaručit, aby byl truhlík vypěstován v nejkratším možném čase a aby byly všechny obsažené druhy výhonků při předání truhlíku zákazníkovi připravené ke sklizni. Systém úkolů by se měl chovat dynamicky, tedy reagovat na čas splnění konkrétního úkolu a na jeho základě naplánovat optimální čas splnění úkolu navazujícího. Navrhovaný systém pro řízení pěstírny by měl pěstírně umožnit efektivně realizovat štíhlou výrobu (lean production). Jedná se o logistický trend, kdy se vyrobí jen tolik produktů, kolik se jich reálně prodá. [66] Metoda štíhlé výroby

je klíčová například tehdy, kdy je omezená skladovatelnost produktů, jako je tomu právě v případě vyprodukovaných truhlíků.

4.2 Návrh aplikace

V rámci návrhu byla navrhovaná aplikace modelována formou diagramů vytvořených pomocí jazyka UML (Unified Modeling Language). UML je standardizovaný jazyk pro modelování softwarových systémů poskytující soubor notací a nástrojů pro vizualizaci, specifikaci a návrh softwarových systémů. [67] Cílem UML je usnadnit porozumění systému a jeho komponentům skrze diagramy, které umožňují modelování různých aspektů systému. Návrh webové aplikace pomocí UML umožňuje detailní specifikaci a vizualizaci celého systému webové aplikace, což zvyšuje jeho přehlednost a umožňuje snadnou identifikaci chyb a nejasností v návrhu systému.

Klíčové procesy v rámci aplikace byly modelovány pomocí jazyka BPMN (Business Process Model and Notation). BPMN je grafická notace používaná pro modelování podnikových procesů. BPMN se využívá k popisu procesů na různých úrovních detailu, od vysokoúrovňového náhledu na celkový proces až po detailní popis jednotlivých kroků konkrétního procesu. [68] V případě návrhu webové aplikace může být BPMN použito pro modelování procesů a toků dat, které jsou součástí webové aplikace. To umožňuje lépe porozumět dílčím procesům a identifikovat možnosti pro zlepšení a optimalizaci těchto procesů. Cílem modelování procesů s využitím BPMN je vytvoření jasného náhledu na celkový proces a jeho jednotlivé kroky. Tento náhled pak může být využit k identifikaci oblastí, které mohou být vylepšeny, ke zvýšení efektivity a kvality procesů a ke snížení nákladů na vývoj a údržbu webové aplikace. Vytvořené diagramy jsou rovněž cennou součástí dokumentace projektu, která usnadňuje další vývoj aplikace a její optimalizaci.

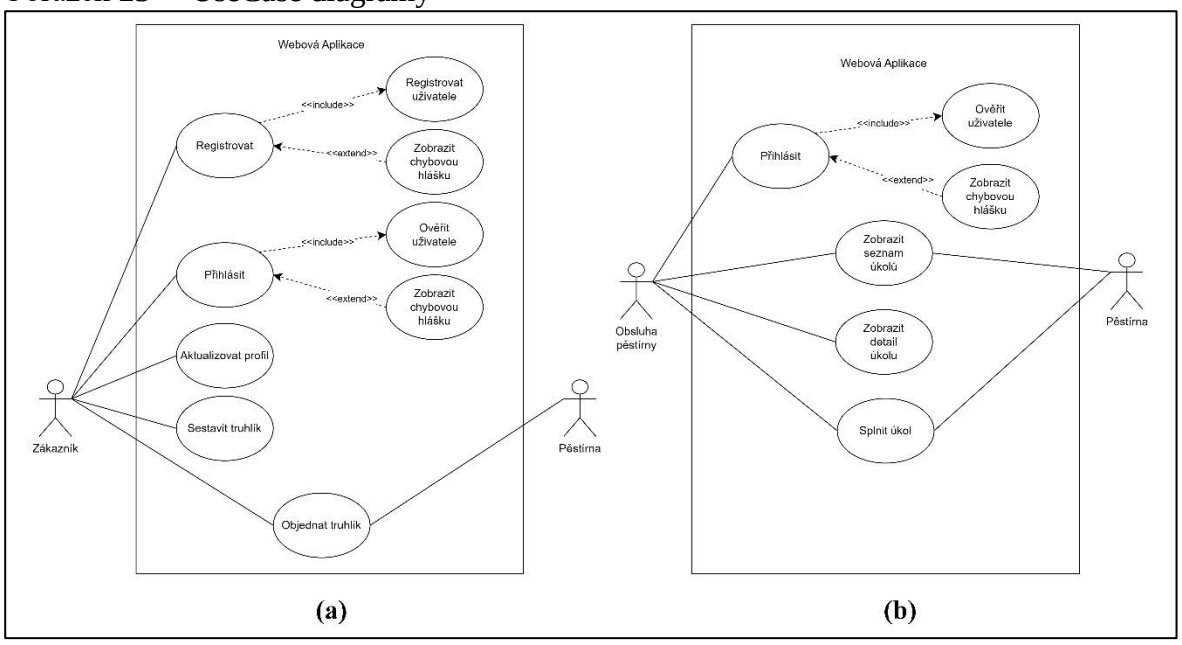
4.2.1 Modelování systému pomocí UML

Chování systému z pohledu uživatelů bylo vymodelováno pomocí UML UseCase diagramů, zobrazených na obrázku 23. Tyto diagramy umožňují specifikovat chování systému z pohledu uživatelů, popsat způsob, jakým bude systém používán a definovat hlavní akce, které bude systém provádět. UseCase diagramy rovněž tvoří základní kámen pro tvorbu dalších podrobnějších diagramů a modelů.

Část (a) obrázku 23 zachycuje UseCase diagram, kde v roli primárního aktéra vystupuje zákazník a v roli sekundárního aktéra pěstírna. Zákazník se do aplikace registruje

nebo se přihlásí ke svému již existujícímu účtu. Po úspěšném přihlášení do aplikace si zákazník sestaví požadovaný truhlík, který si následně objedná. Kdykoliv může také upravit informace o svém profilu, jako je například adresa pro doručení objednávky. V části (b) obrázku 23 je vyobrazen UseCase diagram, kde v roli primárního aktéra vystupuje obsluha pěstírny. Jedná se rovněž o uživatele aplikace, který má ale navíc oprávnění pro spravování pěstírny. Obsluha si po úspěšném přihlášení zobrazí seznam úkolů pro daný den. Dále plní jednotlivé úkoly, v případě potřeby si může prohlédnout detail konkrétního úkolu, kde se obsluze zobrazí detailní informace o zvoleném úkolu.

Obrázek 23 UseCase diagramy

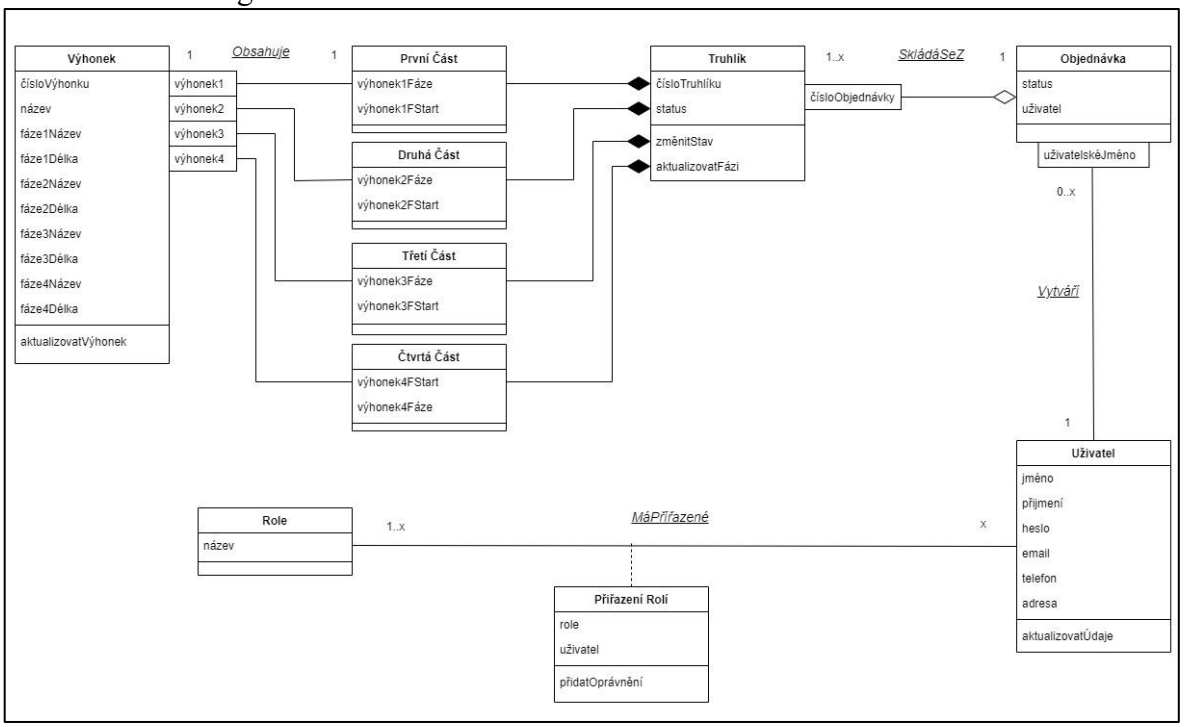


Zdroj: vlastní zpracování

Diagram tříd na obrázku 24 zachycuje třídy tvořící základ aplikace, jejich strukturu a vzájemné vztahy. Na diagramu tříd lze pozorovat třídu uživatel, která má přiřazené příslušné role. Uživatel vytváří objednávku, objednávka se skládá z jednotlivých truhlíků. Každý truhlík se skládá ze 4 částí, kdy každá část obsahuje jeden druh výhonku. Vztah mezi truhlíkem a jeho částmi může být popsán jako kompozice. Jedná se o vztah, kdy samotná část truhlíku nemůže existovat sama o sobě a zároveň jednotlivé části truhlíku mezi sebou nejsou zaměnitelné. To vychází z analýzy požadavků pěstírny popsané v kapitole 4.1 (Analýza požadavků pěstírny), kdy zákazník určuje při sestavování truhlíku zároveň jeho přesné složení. Jinými slovy, pokud zákazník sestaví truhlík, kde je v 1. a 4. části truhlíku slunečnice denní, je potřeba přesně takový truhlík vypěstovat. Pořadí jednotlivých částí v rámci truhlíku je tedy klíčové. Třída výhonek obsahuje informace o výhoncích, jejich

fázích, pořadí fází a délkách trvání jednotlivých fází. Každá část truhlíku obsahuje informaci o výhonku, který v dané části truhlíku roste, aktuální fázi, ve které se rostlinný druh právě nachází a čase, kdy daná fáze započala. Truhlík je pomocí atributu *cisloObjednavky* jednoznačně přiřazen k jedné objednávce. Dále obsahuje atribut *status*, který uchovává informaci o fázi životního cyklu, ve které se truhlík nachází. Objednávka je atributem *uzivatelskeJmeno* přiřazena ke konkrétnímu uživateli. Třída objednávka rovněž obsahuje atribut *status*, uchovávající informaci o fázi životního cyklu objednávky.

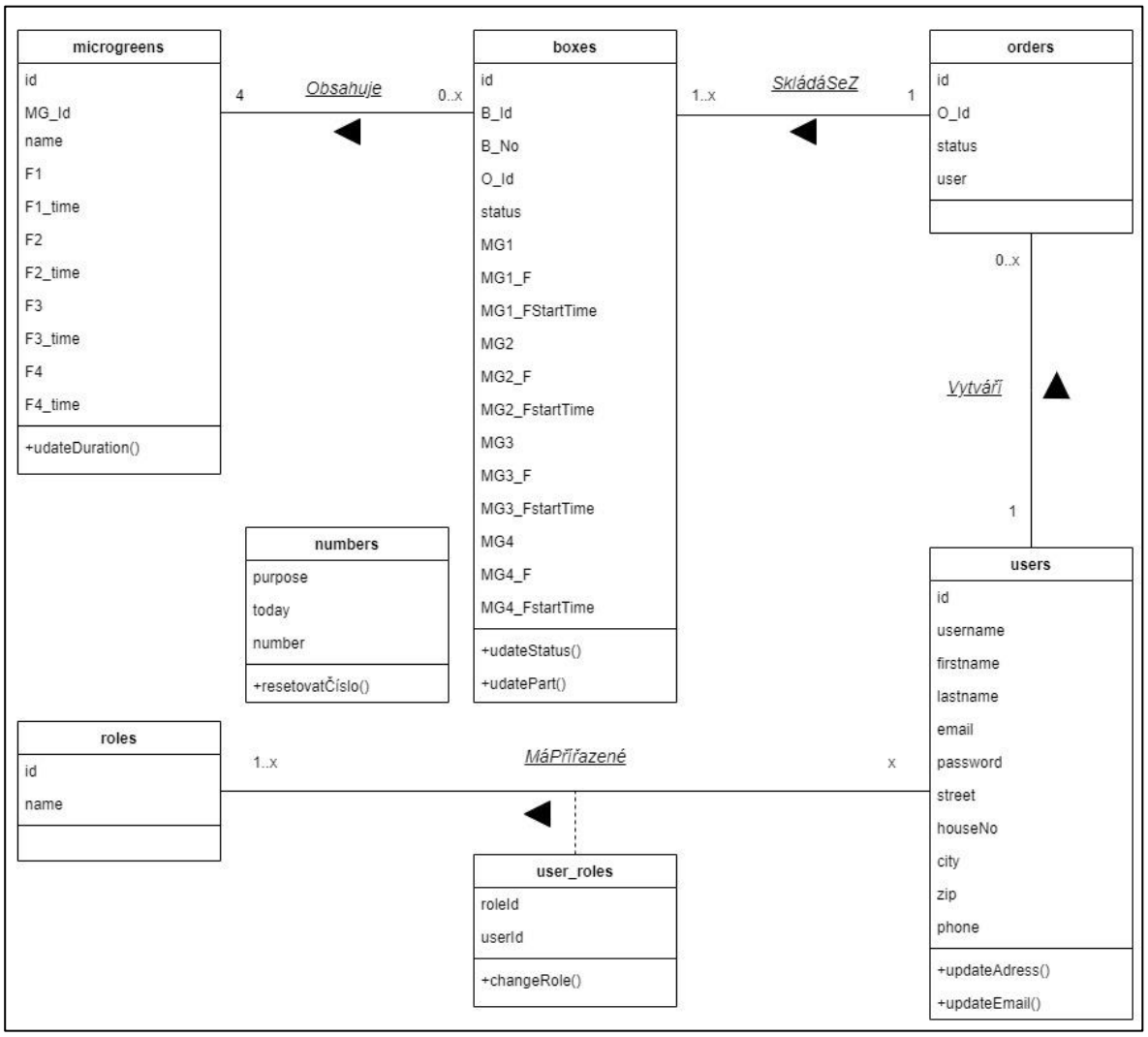
Obrázek 24 Diagram tříd



Zdroj: vlastní zpracování

Na základě diagramu tříd byla vymodelována struktura relační databáze znázorněná na obrázku 25. Třída truhlík a její podtřídy byly v návrhu relační databáze realizovány formou jediné tabulky. To bylo možné díky kompozičnímu vztahu mezi třídou *Truhlík* a jejími podtřídami. Přesto že jsou jednotlivé části truhlíku strukturně totožné, každá z nich má jedinečné umístění v rámci truhlíku, a tedy každá část truhlíku je plně závislá na primárním klíči třídy *Truhlík*. Struktura a názvy všech tříd byly dále upraveny tak, aby splňovaly základní strukturální požadavky relační databáze.

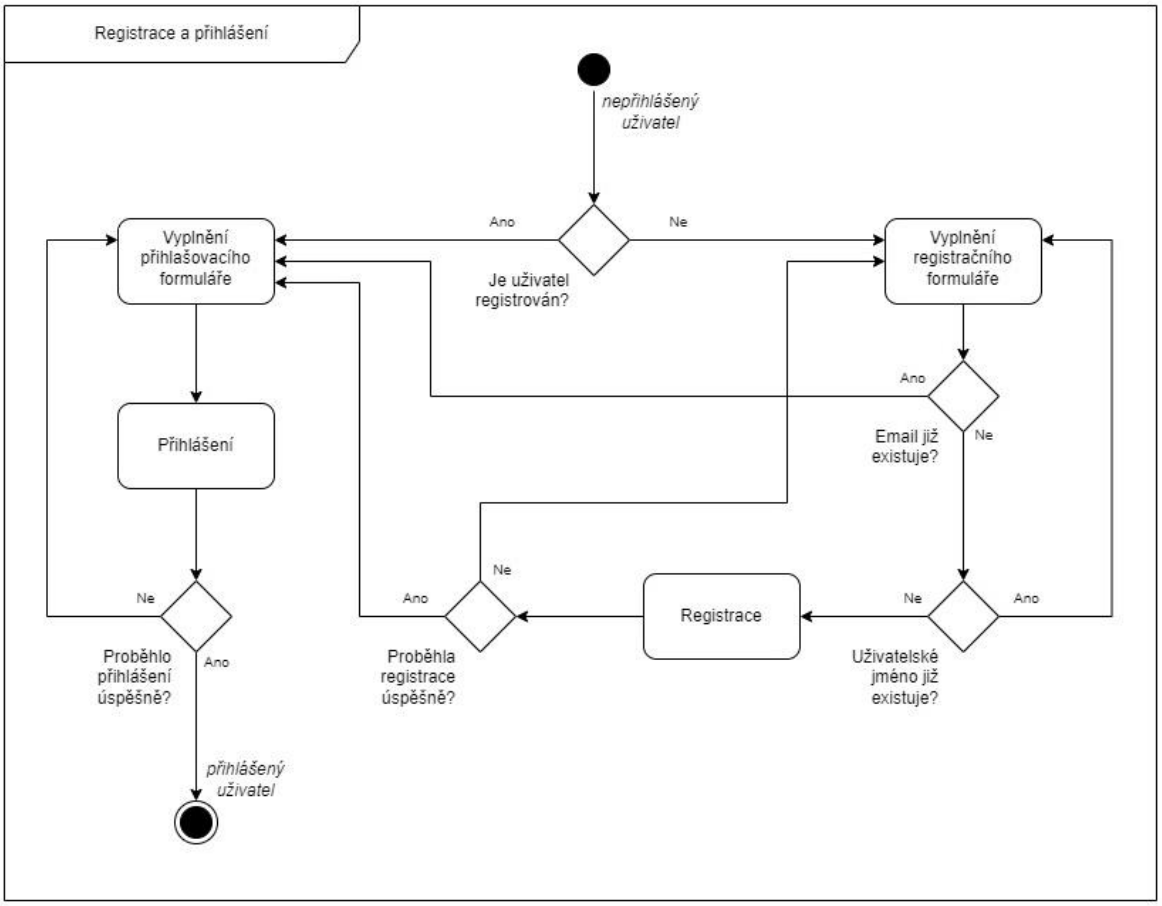
Obrázek 25 Struktura relační databáze



Zdroj: vlastní zpracování

Chování samotného systému lze modelovat pomocí diagramu aktivit. Diagram aktivit poskytuje grafickou reprezentaci toku dat, procesů a rozhodování v systému, což usnadňuje porozumění procesům v systému. Tento diagram může být použit pro modelování procesů webové aplikace, jako je například přihlášení uživatele nebo vytvoření objednávky. V těchto případech umožňuje diagram aktivit vizualizovat jednotlivé kroky, které uživatelé musí provést v rámci dané funkcionality webové aplikace. Diagram aktivit na obrázku 26 zachycuje chování systému při registraci nebo přihlašování uživatele. Na začátku diagramu vystupuje nepřihlášený uživatel. Pokud uživatel není registrován, pokračuje vyplněním registračního formuláře. Pokud uživatel formulář úspěšně vyplní, odešle a registrace proběhne úspěšně, pokračuje uživatel přihlášením do aplikace.

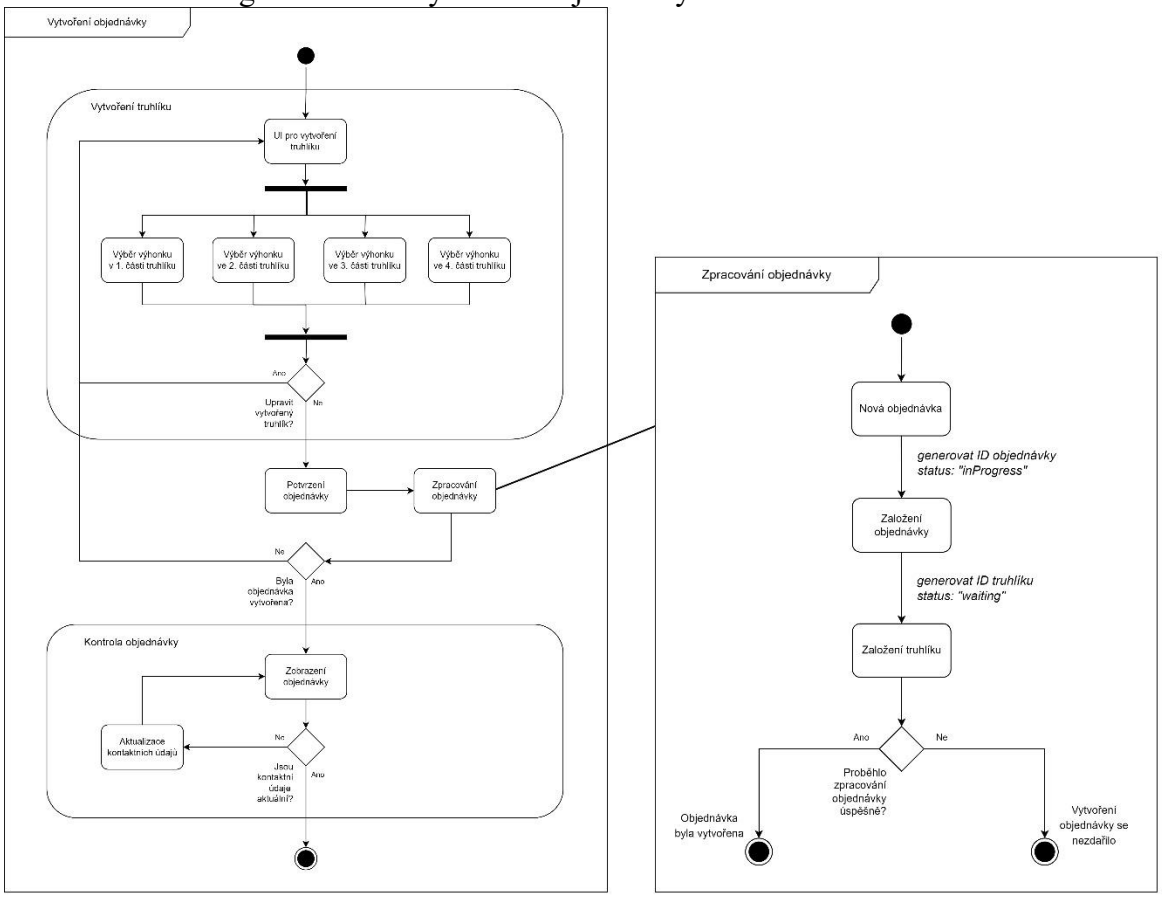
Obrázek 26 Diagram aktivit: registrace a přihlášení



Zdroj: vlastní zpracování

Po úspěšném přihlášení do aplikace si může zákazník sestavit vlastní truhlík a následně vytvořený truhlík objednat. Tento proces zachycuje diagram aktivit na obrázku 27. V části (a) obrázku 27 lze pozorovat sestavení truhlíku zákazníkem, který vybere požadovaný druh výhonku pro každou část sestavovaného truhlíku. Jakmile jsou zaplněny všechny čtyři části truhlíku výhonky a zákazník si nepřeje složení truhlíku dále upravovat, pokračuje potvrzením objednávky, čímž se objednávka odešle ke zpracování. Zpracování objednávky zachycuje část (b) obrázku 27. V průběhu zpracování objednávky dojde k vygenerování unikátního ID objednávky a příslušných truhlíků, k čemuž se využije tabulka *numbers* zachycená na obrázku 25. Vytvořené objednávce a přidruženým truhlíkům se následně nastaví počáteční stav. Pokud vše proběhne úspěšně, zobrazí se uživateli detail uskutečněné objednávky. Uživatel může zpětně upravit adresu doručení nebo pokračovat vytvořením další objednávky.

Obrázek 27 Diagram aktivit: vytvoření objednávky



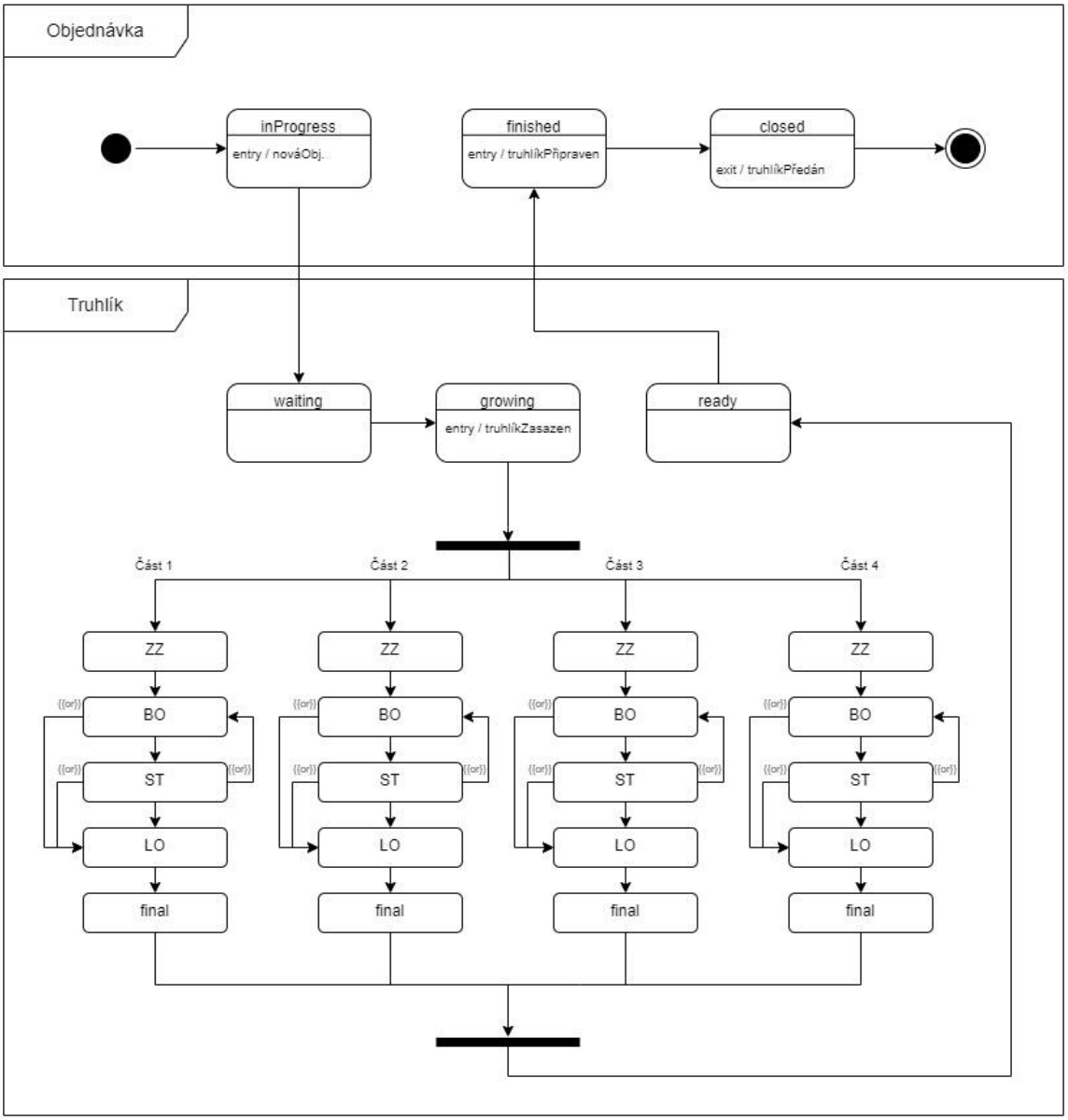
Zdroj: vlastní zpracování

Procesy znázorněné pomocí diagramů aktivit lze vymodelovat pomocí sekvenčního diagramu, který navíc zachycuje časovou posloupnost prováděných aktivit. Sekvenční diagram je vyobrazen v příloze A na obrázku 1. Příloha zachycuje přihlášení uživatele a vytvoření objednávky uživatelem jako komunikaci mezi objekty zákazník, webová aplikace, server a databáze rozloženou v čase.

Při popisu diagramu tříd byla zmíněna problematika stavů objednávky a truhlíků. Během životního cyklu objednávky, tedy od jejího vytvoření až po předání vyprodukovaných truhlíků zákazníkovi, vystřídají objednávka a k ní přidružené truhlíky několik různých stavů. Tyto stavy v případě objednávky reflektují významné fáze jejího zpracování, v případě truhlíku pak fáze pěstebního procesu, kterými truhlík prochází. K identifikaci stavů, ve kterých se objekty mohou nacházet, a přechodů mezi nimi, lze využít stavový diagram. Stavový diagram je typ diagramu, který slouží k modelování chování objektů nebo systému jako automatového stavového diagramu. Stavový diagram na obrázku 28 popisuje stavy objektů objednávka a truhlík. Při přijetí nové objednávky

se objednávka nachází v iniciačním stavu *inProgress* a k ní náležící truhlíky ve stavu *waiting*. Jakmile obsluha pěstírny truhlíku přiřadí číslo reálného truhlíku a vyseje do něj první semena, započne pěstební proces a truhlík přechází do stavu *growing*. V průběhu pěstebního procesu jsou sledovány samostatně všechny čtyři části truhlíku, ve kterých jednotlivé zaseté druhy postupně střídají své růstové fáze. Jakmile výhonky ve všech částech truhlíku dospějí do fáze *final* (což značí, že jsou připravené ke sklizni), přechází truhlík do stavu *ready* a objednávka do stavu *finished*. Po předání truhlíku zákazníkovi přechází objednávka do terminálního stavu *closed*.

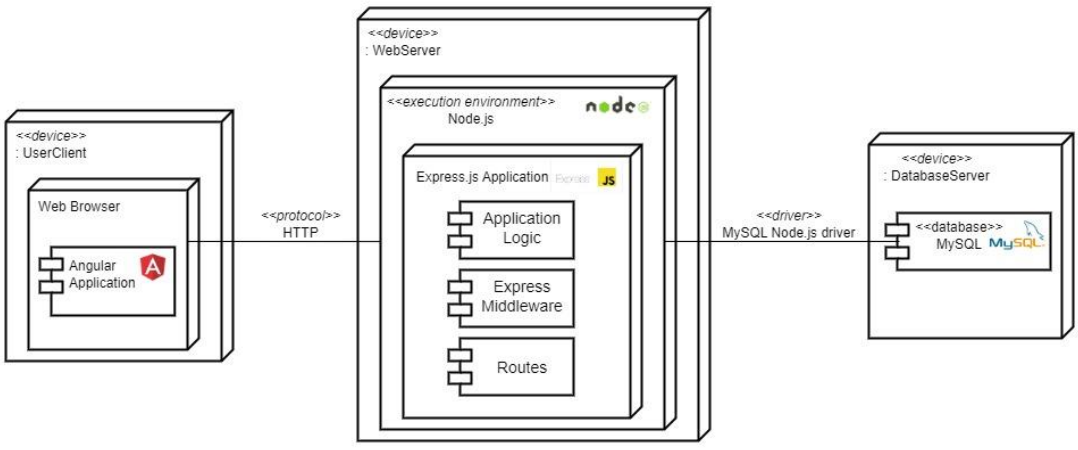
Obrázek 28 Stavový diagram



Zdroj: vlastní zpracování

K modelování fyzického rozložení systému webové aplikace lze využít diagram nasazení. Diagram nasazení ukazuje, jak jsou jednotlivé komponenty aplikace distribuovány a jak spolu komunikují. Obsahuje specifikaci hardwarových a softwarových komponentů, které jsou nutné pro běh aplikace. Důležitým prvkem diagramu nasazení webové aplikace je zobrazení síťových propojení mezi jednotlivými komponenty. To zahrnuje propojení mezi serverem a klientem a komunikaci mezi databází a serverovou částí aplikace. Diagram nasazení navrhované aplikace je zachycen na obrázku 29. Diagram rozděluje aplikaci na 3 části. První částí je uživatelské zařízení s webovým prohlížečem, ve kterém uživatel zobrazuje klientskou část webové aplikace, popsanou v kapitole 3.1 (Framework Angular). Klientská část aplikace komunikuje prostřednictvím internetu s využitím protokolu HTTP se serverovou částí aplikace, která se nachází na webovém serveru (backend) a kterou popisuje kapitola 3.5 (Serverová část aplikace). Na fyzickém serveru se nachází serverové prostředí Node.js spolu s aplikační vrstvou Express.js. Třetí částí aplikace je databázový server s databázovým systémem MySQL nacházející se na stejném fyzickém zařízení jako serverová část aplikace. Serverová část aplikace komunikuje s databází pomocí MySQL Node.js ovladače.

Obrázek 29 Diagram nasazení



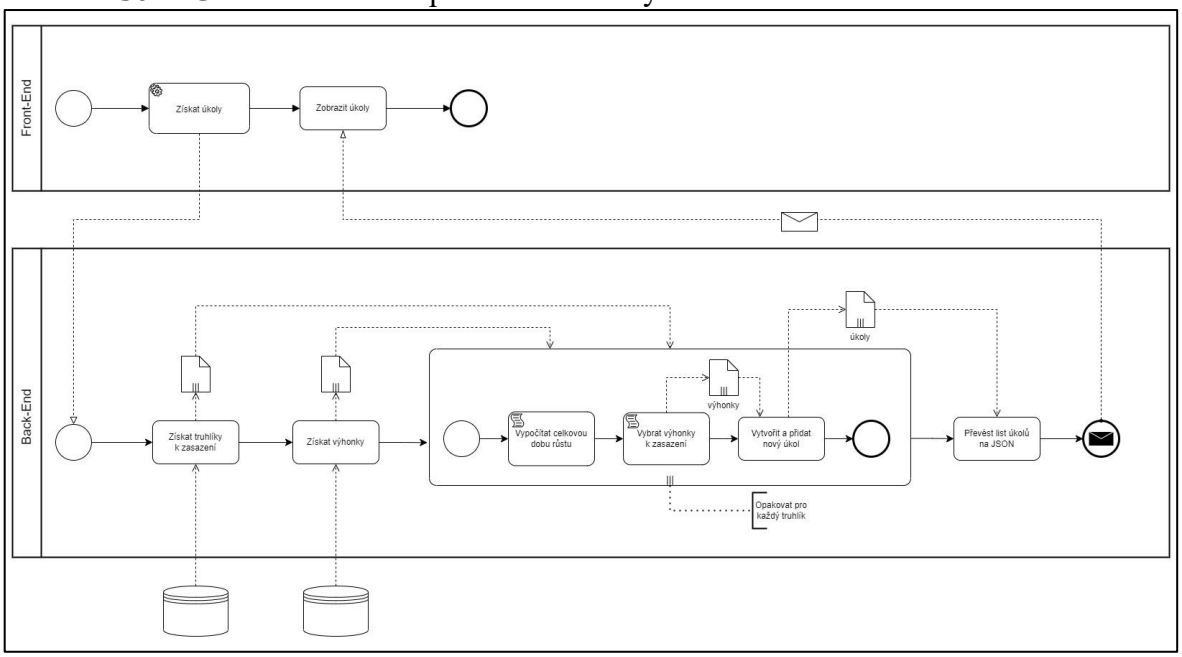
Zdroj: vlastní zpracování

4.2.2 Modelování procesů pomocí BPMN

V rámci návrhu aplikace byly pomocí BPMN modelovány klíčové procesy zajišťující chod pěstírny, a to formou spolupráce mezi klientskou a serverovou částí webové aplikace. Úkoly prováděné obsluhou pěstírny lze rozdělit do tří kategorií: zakládání nových truhlíků, setí semen a manipulační úkoly. Diagram aktivit na obrázku 27 popisuje proces

vytvoření objednávky zákazníkem. Truhlíky vzniklé přijetím objednávky se na konci tohoto procesu nacházejí ve stavu *waiting*, jak lze pozorovat na stavovém diagramu na obrázku 28. Prvním typem úkolu pro obsluhu pěstírny je zasetí semen nejdéle rostoucích výhonků do patřičných částí truhlíku a zadání čísla použitého reálného truhlíku do systému. Tím dojde v systému aplikace k založení nového truhlíku a odstartování jeho pěstebního procesu. Proces generování těchto úkolů znázorňuje diagram na obrázku 30. Při vstupu přihlášeného uživatele s příslušným oprávněním do sekce *Pěstírna* se klientská část aplikace dotáže pomocí příslušného API volání serverové části aplikace na seznam úkolů pro zakládání nových truhlíků. Příslušná funkce serverové části aplikace nejdříve získá z databáze všechny truhlíky z tabulky *boxes*, které čekají na založení – tedy nacházejí se ve stavu *waiting*. Následně jsou z databáze získány informace o všech výhoncích, které se v pěstírně pěstují. Pro každý nalezený truhlík dojde k identifikaci obsažených výhonků v jednotlivých částech truhlíku a následnému výpočtu celkové doby potřebné pro vypěstování daného truhlíku. Na základě celkové doby potřebné pro vypěstování truhlíku se vyberou výhonky, které je potřeba zasít jako první. Z informací o příslušném druhu výhonku a části truhlíku, do které je potřeba semena výhonku zasít, je následně vytvořen úkol, který se přidá do seznamu úkolů. Jakmile tento proces proběhne pro každý nalezený truhlík a seznam úkolů je tudíž kompletní, převede se seznam úkolů do formátu JSON a je odeslán uživateli zpět do klientské části aplikace. Zdrojový kód funkce generující tento typ úkolů je v ukázce kódu 1 v příloze B.

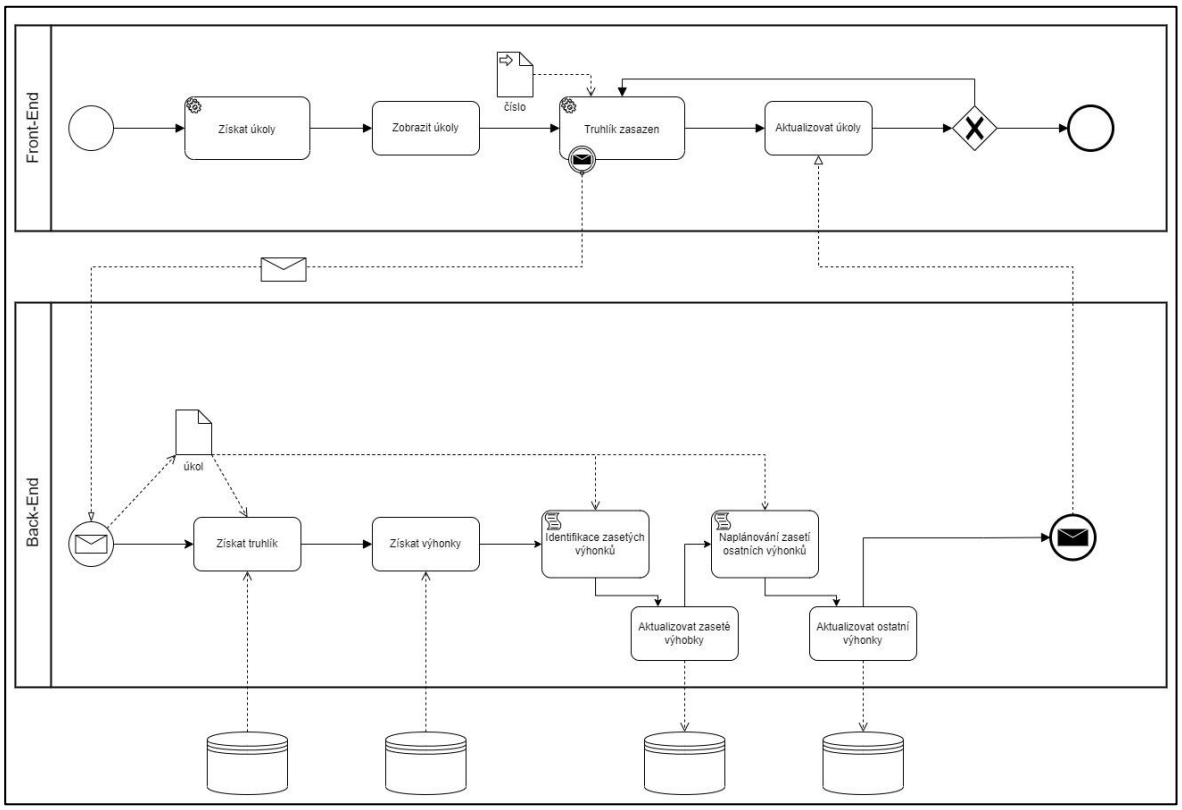
Obrázek 30 Generování úkolů pro založení nových truhlíků



Zdroj: vlastní zpracování

Po zobrazení seznamu úkolů může uživatel přejít k plnění jednotlivých úkolů. Proces plnění úkolu je vymodelován na obrázku 31. Pro splnění úkolu musí obsluha pěstírny zasít semena příslušných výhonků do požadovaných částí truhlíku a zadat číslo reálného truhlíku, do kterého byla semena vyseta. Poté může odkliknout splnění úkolu. Při potvrzení splnění úkolu se pomocí příslušného API volání odešlou serverové části aplikace informace o splnění úkolu včetně čísla reálného truhlíku, který byl použit pro výsadbu. Serverová část aplikace nejdříve získá z databáze příslušný truhlík spolu s informacemi o všech výhoncích pěstovaných v pěstírně. Následně dojde k identifikaci zasazených výhonků v odpovídajících částech truhlíku a následné aktualizaci truhlíku v databázi. Na základě času splnění úkolu jsou dále vypočítány optimální časy splnění dalších úkolů týkajících se daného truhlíku, ostatní části truhlíku v databázi jsou aktualizovány a truhlík přechází do stavu *growing*. Do klientské části aplikace je odeslána informace o úspěšném zaznamenání splnění úkolu, na což klientská část aplikace reaguje aktualizací seznamu úkolů. Uživatel může následně pokračovat plněním dalších úkolů. Zdrojový kód funkce zpracovávající splnění úkolu je vyobrazen pomocí ukázek kódu 1,2,3 a 4 v příloze C.

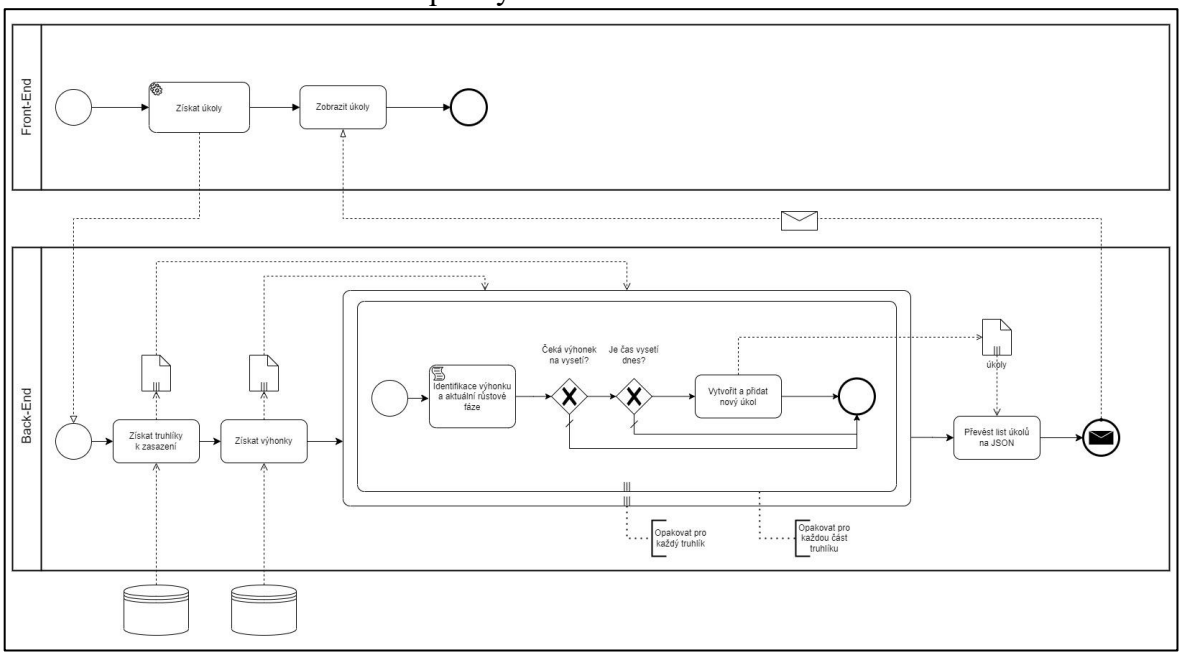
Obrázek 31 Založení nového truhlíku



Zdroj: vlastní zpracování

Dalším typem úkolu, který bude obsluha pěstírny vykonávat, je vysetí semen nového druhu výhonku do požadované části již založeného truhlíku. Generování úkolů tohoto typu znázorňuje diagram na obrázku 32. Při vstupu přihlášeného uživatele s příslušným oprávněním do sekce *Pěstírna* je realizováno API volání dotazující se na seznam úkolů pro vysetí semen nových výhonků do již existujících truhlíků. Příslušná funkce serverové části aplikace nejdříve získá z databáze všechny truhlíky z tabulky *boxes*, které se nacházejí ve stavu *growing*. Následně jsou získány z databáze informace o všech výhoncích, které se v pěstírně pěstují. Poté proběhne pro každou část každého nalezeného truhlíku identifikace druhu výhonku. Pokud se výhonek ve zkoumané části truhlíku nachází ve stavu *ZZ*, tedy čeká na vysetí, a pokud je naplánovaný čas vysetí právě dnes, dojde k vytvoření úkolu, který je přidán do seznamu úkolů. Jakmile dojde k analýze všech částí všech truhlíků nacházejících se ve stavu *growing*, a tedy seznam úkolů je kompletní, dojde k převedení seznamu úkolů na příslušný formát a odeslání seznamu úkolů do klientské části aplikace, kde je seznam úkolů zobrazen obsluze pěstírny. Seznam úkolů obsahuje číslo reálného truhlíku, část truhlíku, druh výhonku k vysetí a optimální naplánovaný čas splnění úkolu.

Obrázek 32 Generování úkolu pro výsev semen

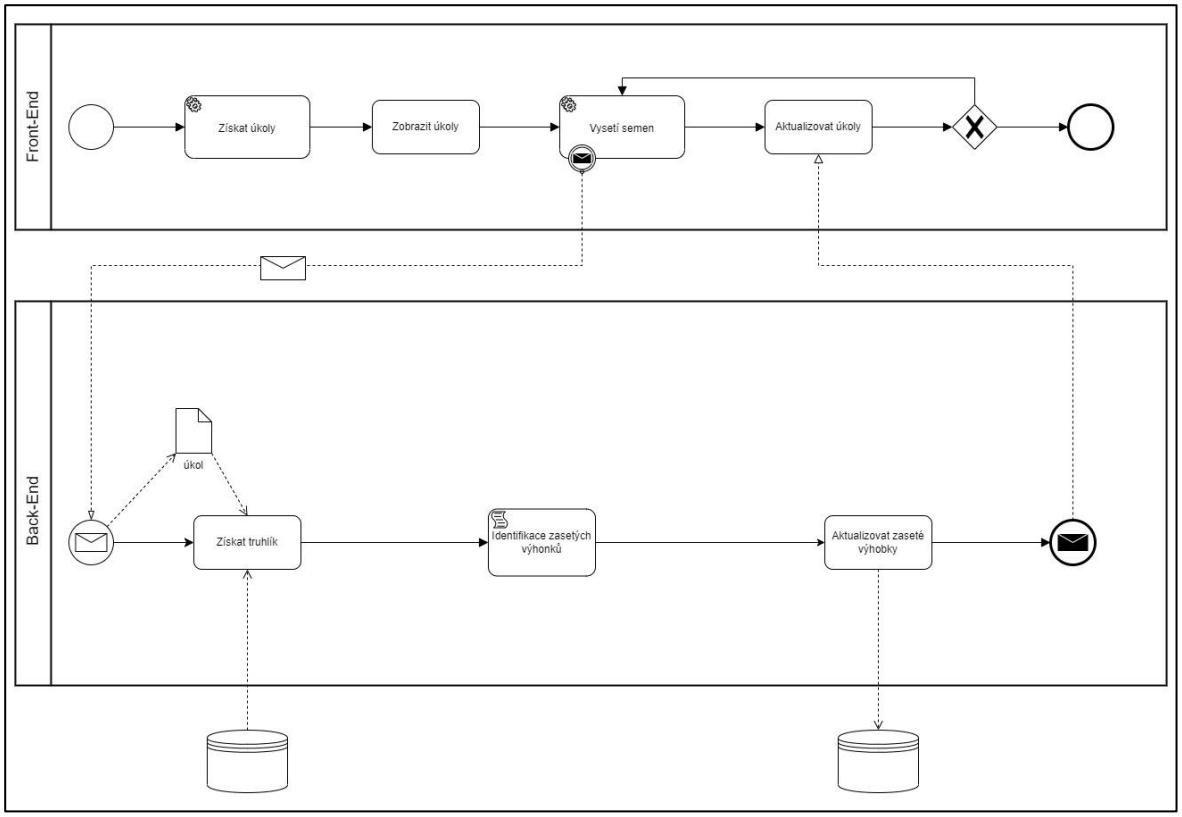


Zdroj: vlastní zpracování

Po zobrazení seznamu úkolů týkajících se výsevu semen pokračuje obsluha pěstírny plněním jednotlivých úkolů. Pro splnění úkolu musí obsluha pěstírny v pěstírně vyhledat příslušný truhlík podle čísla reálného truhlíku zadaného při založení truhlíku a do určené

části truhlíku zasít semena požadovaného druhu výhonku. Poté v klientské části aplikace potvrdí splnění úkolu. Splnění úkolu je zaznamenáno pomocí procesu vyobrazeném na obrázku 33. Pomocí API volání je informace o splněném konkrétního úkolu odeslána do serverové části aplikace. Funkce zpracovávající splněný úkol načte z databáze informace o příslušném truhlíku, identifikuje provedené změny a zapíše je zpět do databáze. Klientská část aplikace reaguje na úspěšné splnění úkolu aktualizováním seznamu úkolů.

Obrázek 33 Setí semen

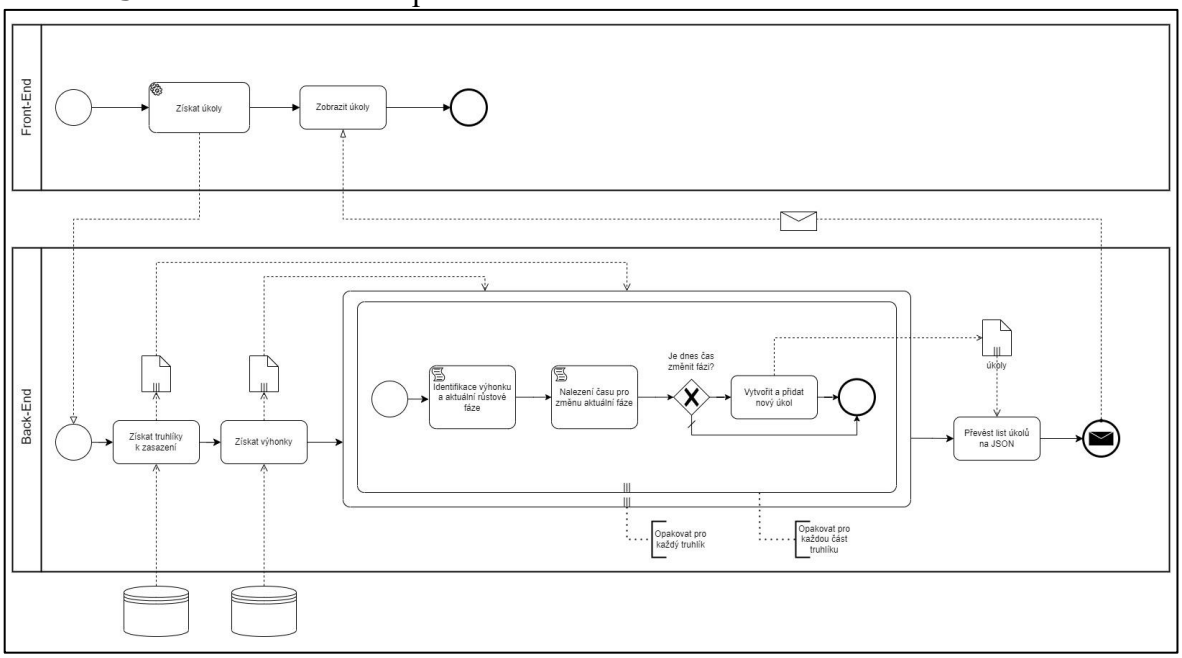


Zdroj: vlastní zpracování

Posledním typem úkolů, které bude obsluha pěstírny provádět, jsou úkoly manipulační. Jedná se o takové manipulace s truhlíky, které umožňují výhonkům v jednotlivých částech truhlíku přecházet do navazujících růstových fází. Může se jednat o zatížení klíčků, čímž klíčky přejdou do fáze vršení, nebo odkrytí klíčků, čímž se klíčky vystaví světlu a přejdou do fáze růstu. Proces generování manipulačních úkolů je zachycen na obrázku 34. Při vstupu obsluhy pěstírny do sekce *Pěstírna* v klientské části aplikace je realizováno API volání dotazující se na seznam manipulačních úkolů. Příslušná funkce serverové části aplikace nejdříve získá z databáze všechny truhlíky z tabulky *boxes*, které se nacházejí ve stavu *growing*. Dále jsou získány z databáze informace o všech výhoncích

pěstovaných v pěstírně. Následně proběhne pro každou část každého nalezeného truhlíku identifikace druhu výhonku a růstové fáze, ve kterém se výhonek právě nachází. Na základě času, kdy aktuální fáze růstu započala, je vypočítáno datum a čas, kdy by měl příslušný výhonek přejít do navazující fáze. Pokud je vypočtené datum pro změnu fáze shodné s dnešním datem, je vytvořen úkol, který je následně přidán do seznamu manipulačních úkolů. Jakmile proběhne popsaný cyklus pro všechny části všech nalezených truhlíků, seznam úkolů je převeden do vhodného formátu a odeslán do klientské části aplikace, kde je následně zobrazen obsluze. Seznam manipulačních úkolů obsahuje pro každý úkol informaci o čísle reálného truhlíku, cílové části truhlíku a výhonku, který v dané části roste, spolu s informací o aktuální fázi růstu, navazující fázi růstu a optimálním čase, kdy by měl být přechod mezi fázemi realizován.

Obrázek 34 Generování manipulačních úkolů

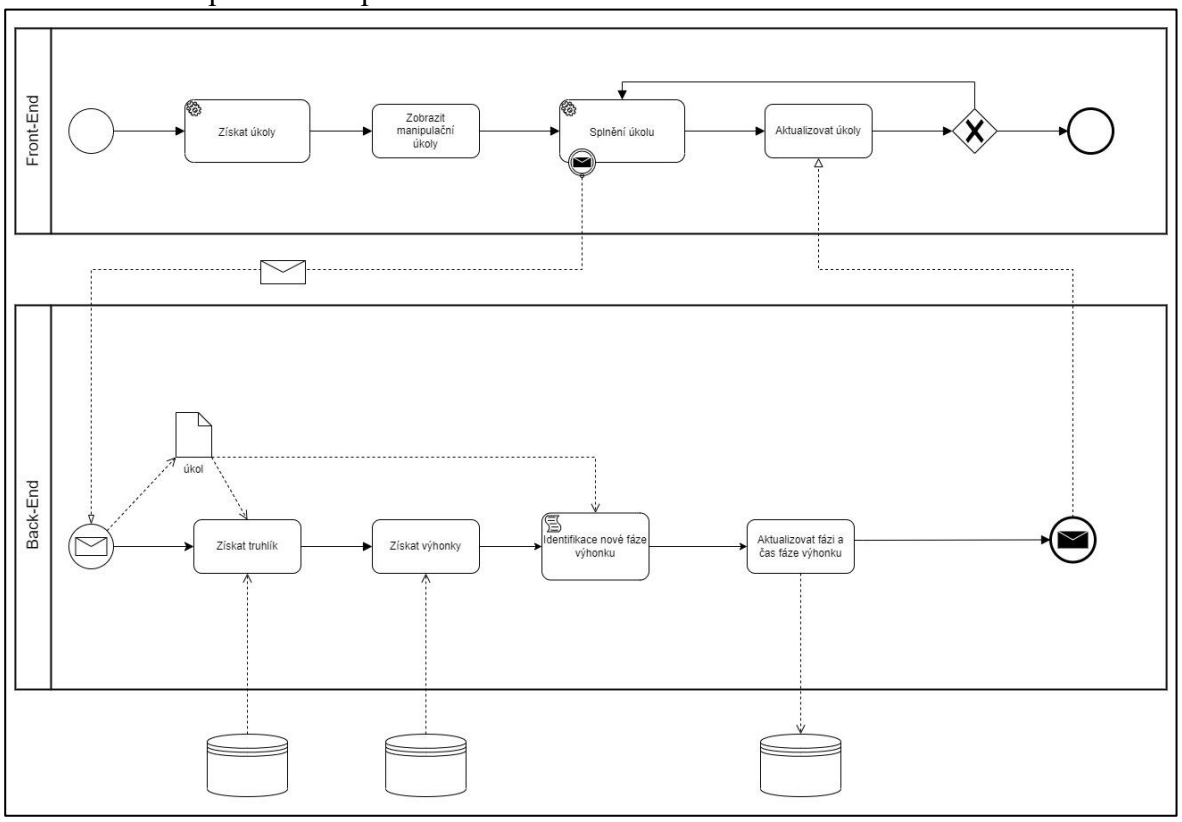


Zdroj: vlastní zpracování

Jakmile provede obsluha pěstírny příslušný manipulační úkol, čímž přejde konkrétní druh výhonku v dané části truhlíku do další růstové fáze, potvrdí obsluha pěstírny splnění úkolu v klientské části aplikace. Informace o splněném úkolu jsou prostřednictvím API volání odeslány do serverové části aplikace, kde jsou následně zpracovány příslušnou funkcí. Funkce serverové části aplikace vyhledá v databázi příslušný truhlík a identifikuje druh výhonku rostoucí v dané části truhlíku. Na základě aktuální růstové fáze výhonku je identifikována následující fáze růstu, do které výhonek splněním úkolu přešel. Nová fáze růstu výhonku je spolu s časem započetí této fáze zaznamenána do databáze. Informace

o úspěšném zaznamenání splnění úkolu je odeslána zpět do klientské části aplikace, která reaguje aktualizací seznamu manipulačních úkolů. Proces zaznamenání splněného manipulačního úkolu je zachycen na obrázku 35.

Obrázek 35 Splnění manipulačního úkolu

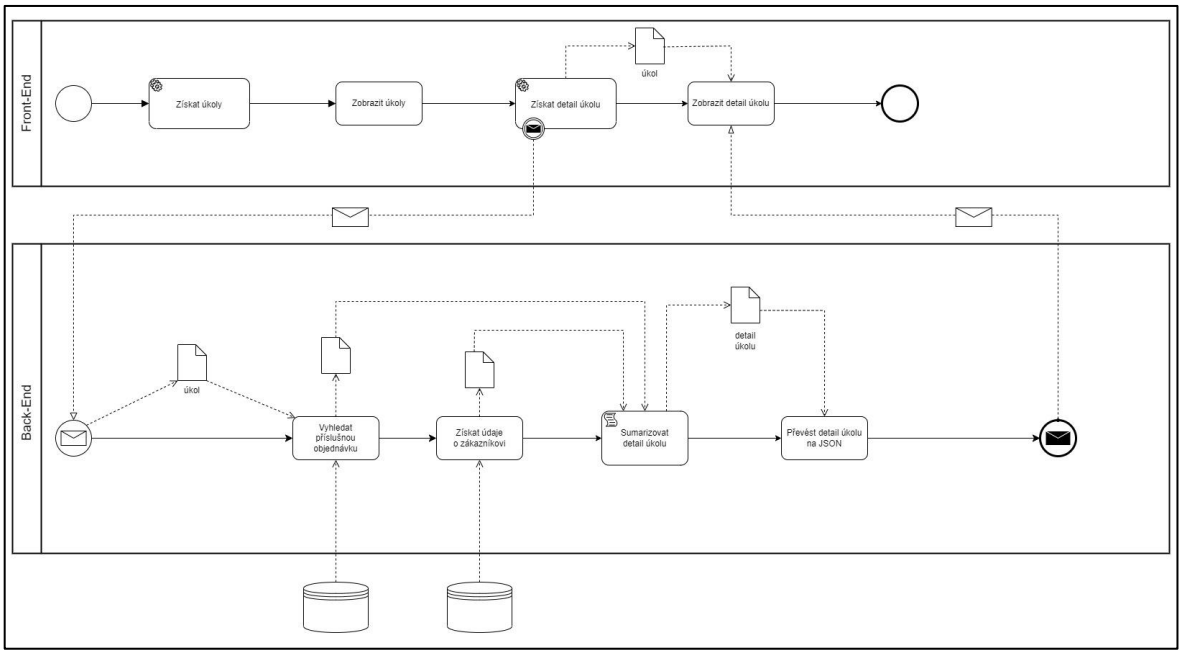


Zdroj: vlastní zpracování

V rámci plnění úkolů může nastat situace, kdy obsluze pěstírny nebudou údaje o úkolu zobrazované v rámci seznamů úkolů dostačovat. V takovém případě může obsluha pěstírny zobrazit detail úkolu obsahující rozšířené informace o daném úkolu, jako jsou ID truhlíku a objednávky, jméno zákazníka nebo datum vytvoření objednávky. Detail konkrétního úkolu může obsluha pěstírny zobrazit kliknutím na příslušnou ikonu, následkem čehož se klientská část aplikace pomocí API volání dotáže na doplňující informace k úkolu. Funkce zpracovávající API volání v serverové části aplikace přijme informace o úkolu, ke kterému je potřeba dohledat údaje. Na základě těchto informací je vyhledána objednávka, ke které je přiřazen truhlík, kterého se prováděný úkol týká. Následně jsou z databáze načteny údaje zákazníka, který objednávku vytvořil. Načtená data jsou sumarizována, převedena do příslušného formátu a odeslána zpět do klientské části aplikace, kde se spolu

s daty obsaženými v seznamu úkolů uspořádají do layoutu pro detail úkolu a zobrazí se uživateli. Proces zobrazení detailu úkolu je vymodelován na obrázku 36.

Obrázek 36 Zobrazení detailu úkolu



Zdroj: vlastní zpracování

4.3 Vývoj aplikace

Navržený systém byl implementován formou webové aplikace. Vývoj aplikace byl rozdělen na několik fází. V první fázi vývoje byla vytvořena webová aplikace umožňující autentizaci uživatelů a poskytování obsahu na základě autorizace přihlášeného uživatele. Tato aplikace byla v další fázi vývoje rozšířena o zákaznickou část, ve které mohou zákazníci sestavovat vlastní truhlíky a vytvářet objednávky. V poslední fázi vývoje byla aplikace rozšířena o část umožňující správu pěstírny prostřednictvím úkolů pro obsluhu pěstírny. Následující kapitoly zachycují jednotlivé fáze vývoje aplikace.

4.3.1 Autentizace a autorizace uživatelů

Prvním krokem implementace navrženého systému bylo vytvoření serverové části aplikace umožňující autentizaci uživatelů a jejich následnou autorizaci. Serverová část aplikace využívá prostředí *Node.js v16.14.2* a *Express v4.18.2* spolu s *ORM Sequelize v6.25.3* pro interagování s databázovým systémem MySQL. Klientská a serverová část aplikace spolu komunikují prostřednictvím HTTP pomocí služby *HttpClient* podle architektury RestAPI. Zabezpečená komunikace mezi oběma stranami je zajištěna využitím technologie *JsonWebToken v8.5.1*, pomocí které je generován token dále uchovávaný v prohlížeči uživatele, což umožňuje middleware *cookie-session v2.0.0*. Zmíněné technologie a problematika zpracování HTTP požadavku jsou detailně popsány v kapitole 3.5.2 (Komunikace prostřednictvím HTTP).

Serverové prostředí bylo inicializováno v souboru *server.js* pomocí programovacího jazyka JavaScript. Tento soubor je vyobrazen v ukázce kódu 34. Ukázka zachycuje import modulů pro realizaci RestAPI, *cookie-session* pro ukládání dat probíhající relace na straně klienta a CORS k řízení přístupů. Následně jsou moduly nakonfigurovány a jako formát pro přijímané dotazy je nastaven formát JSON. Nakonec je nastaven port, na kterém bude serverová část aplikace naslouchat příchozím připojením.

Ukázka kódu 34 server.js

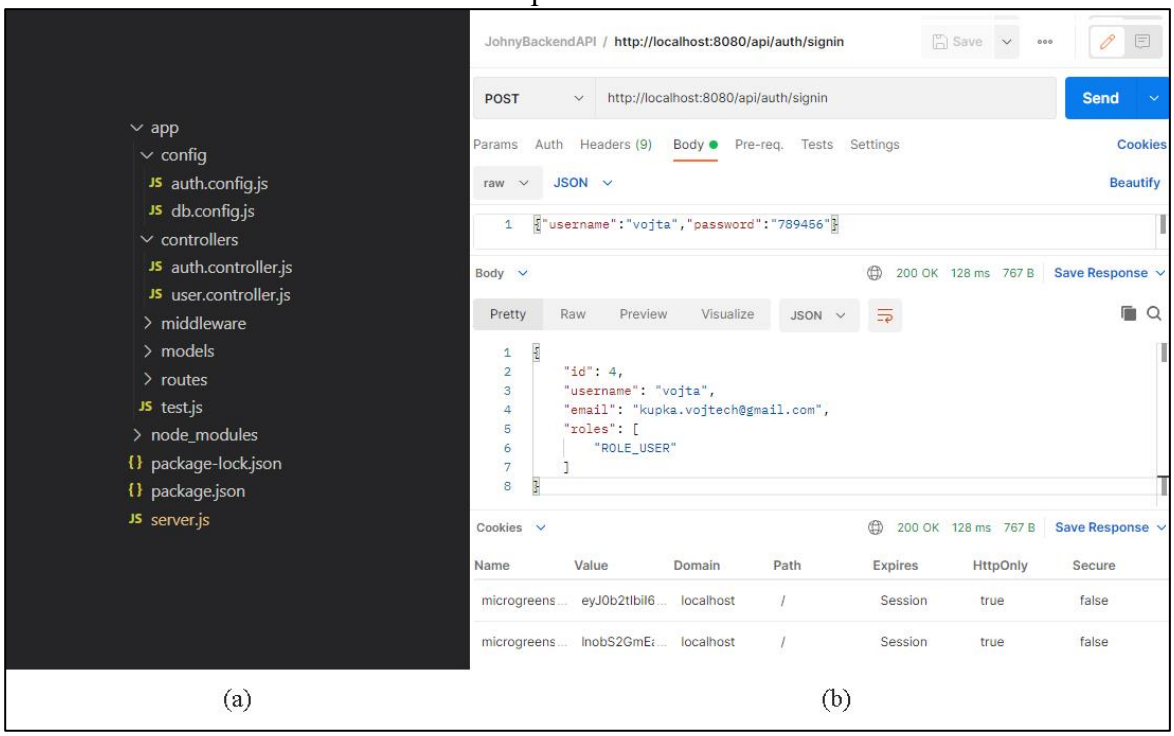
```
// Import modulu
const express = require("express");
const cors = require("cors");
const cookieSession = require("cookie-session");
const app = express();

// Nastaveni CORS
app.use(
  cors({
    credentials: true,
    origin: ["http://localhost:8081"],
  })
);

// Nastaveni formatu pozadavku
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Nastaveni cookie-session
app.use(
  cookieSession({
    name: "microgreens-session",
    secret: "SECRET",
    httpOnly: true
  })
);

// Nastaveni portu pro prichozi pozadavky
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}.`);
});
```

Zdroj: vlastní zpracování

Serverová část aplikace musí dále obsahovat konfigurační soubory *db.config.js* a *auth.config.js* pro konfiguraci připojení Sequelize k MySQL databázi a nastavení řetězce, na jehož základě dochází ke generování JWT. V souborech bezpečnostní vrstvy je implementována logika pro ověřování uživatele a v souborech routeru jsou definovány cesty k funkcím kontroléru. Kontrolér je složen ze souboru *auth.controller.js* implementujícího programovou logiku pro registraci a přihlášení uživatelů a souboru *user.controller.js* obsahujícího funkce implementující ostatní aplikační logiku. Poslední klíčovou součástí serveru jsou modely, ve kterých je definována struktura databáze, jednotlivé tabulky a jejich vzájemné vztahy. Struktura serverové části aplikace je zachycena v části (a) obrázku 37. V části (b) obrázku 37 je zachycena komunikace se serverem při přihlašování uživatele z pohledu klienta. Je zde realizováno příslušné API volání, v rámci kterého jsou v těle dotazu na server odeslány zadané uživatelské jméno a heslo. Po úspěšném ověření přihlašovacích údajů byla navrácena odpověď obsahující informace o přihlášeném uživateli spolu s vytvořeným JWT obsaženém v souboru cookies. Aplikace rozlišuje tři uživatelské role: *user*, *moderator* a *administrator*. Roli *user* má přiřazenou zákazník,

jako je tomu v tomto případě. Roli *moderator* má přiřazenou obsluha pěstírny a role *administrator* je vyhrazena pro vývoj a údržbu aplikace. Na základě role uživatele je uživateli dále zobrazován patřičný obsah.

Obrázek 37 Struktura serverové části aplikace a realizace HTTP dotazu



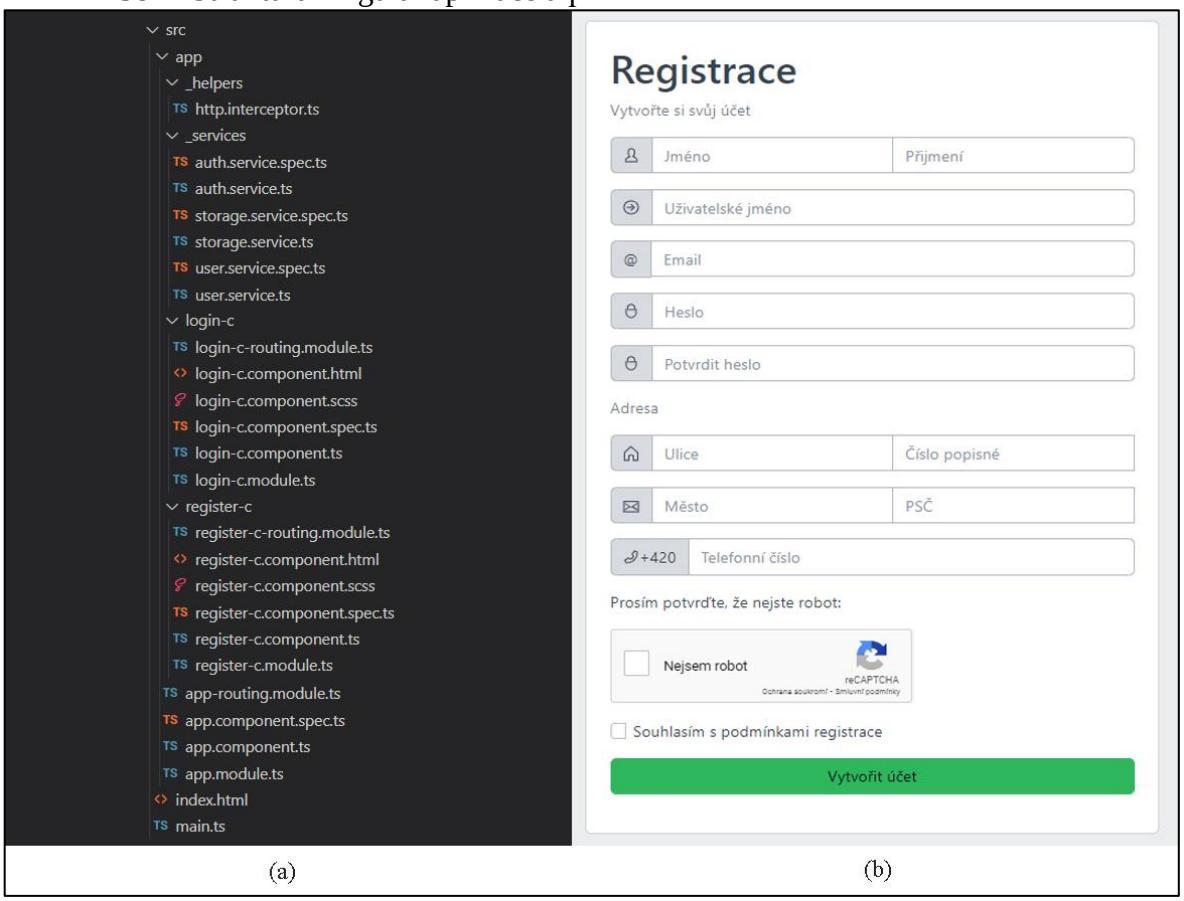
Zdroj: vlastní zpracování

Klientská část aplikace byla realizována pomocí frameworku *Angular 13*, projekt aplikace byl spravován pomocí nástroje *Angular CLI v13*. Součástí klientské části aplikace je knihovna *RxJS v7* pro asynchronní programování umožňující zpracovávání dat z backendu. Pro tvorbu UI byl použit open-source administrační template *CoreUI Admin* pro Angular. Použitý template obsahuje responzivní layout pro mobilní a desktopová zařízení a škálu připravených komponentů, jako jsou tlačítka, formuláře nebo tabulky. Šablona je postavena na frameworku *Bootstrap 4*, komponenty šablony lze upravovat pomocí CSS. V neposlední řadě šablona nabízí flexibilní strukturu pro vytváření vlastních komponentů. Angular aplikace byla vytvořena v programovacím jazyce TypeScript. Problematikou Angular aplikací a jejich vývojem se zabývá kapitola 3.2 (Architektura Angular aplikace) a kapitoly navazující.

Jak bylo dříve řečeno, Angular aplikace je složena z komponentů. Kořenovým komponentem je *app.component* obsahující Angular Router, který mapuje URL adresy na různé komponenty aplikace čímž spravuje navigaci mezi jejími částmi. Kořenový

komponent získává data o uživateli z uložiště prohlížeče BSS (Browser Session Storage) prostřednictvím služby *storage.service*. Aplikace využívá služby *auth.service* implementující službu *HttpClient* pro přihlášení uživatele. Každý požadavek přihlášeného uživatele je poté před odesláním transformován pomocí interceptoru *http.interceptor.ts*, který do hlavičky požadavku vkládá JWT. Metody potřebné pro získávání veřejného a chráněného obsahu jsou poskytovány službou *user.service*. Pro registraci a přihlášení uživatele byly vytvořeny komponenty *register.component* a *login.component* obsahující formuláře pro registraci a přihlášení. Na obrázku 38 v části (a) jsou vyobrazeny popsané komponenty a služby v rámci struktury vytvořené Angular aplikace. V části (b) obrázku 38 je zachycen formulář pro registraci nového uživatele. Registrační formulář obsahuje kompletní údaje o uživateli potřebné pro jeho registraci. Vstupy registračního formuláře jsou validovány v rámci UI pomocí validace formulářů, součástí formuláře je *Google reCAPTCHA* sloužící jako ochrana proti robotům.

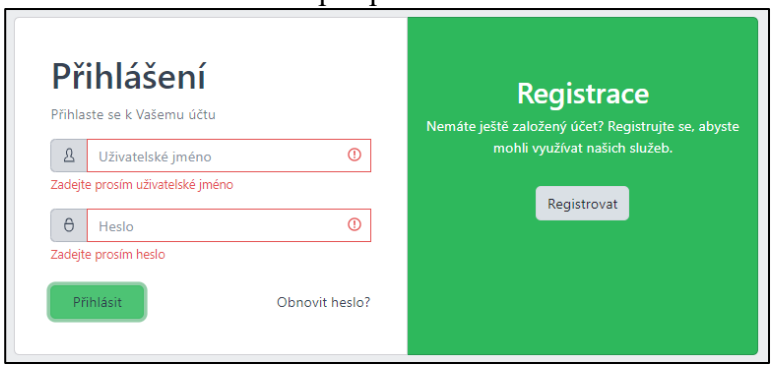
Obrázek 38 Struktura Angular aplikace a přihlašovací formulář



Zdroj: vlastní zpracování

Registrovaný uživatel se do aplikace přihlásí zadáním jména a hesla a následným potvrzením přihlašovacího formuláře zobrazeného na obrázku 39. Formulář je po potvrzení validován na straně klienta a po úspěšné validaci je zavolána funkce *onSubmit()* zachycená v ukázce kódu 35, která je implementována ve třídě komponentu *login.component*. Funkce *onSubmit()* prostřednictvím služby *auth.service* s využitím reaktivního programování realizuje HTTP požadavek pro přihlášení uživatele. Služba *auth.service* realizuje příslušné API volání pomocí funkce *login()*, která je zobrazena v ukázce kódu 36.

Obrázek 39 Formulář pro přihlášení uživatele



Zdroj: vlastní zpracování

Ukázka kódu 35 login.component: onSubmit()

```
onSubmit(): void {
    const { username, password } = this.form;
    this.authService.login(username, password).subscribe({
      next: data => {
        this.storageService.saveUser(data);
        this.roles = this.storageService.getUser().roles;
        setTimeout(() => {
          this.router.navigate(['/', 'home'])
            .then(()=>{
              window.location.reload();
            });
        }, 2000)
      },
      error: err => {
        this.errorMessage = err.error.message;
      }
    });
  }
```

Zdroj: vlastní zpracování

Ukázka kódu 36 auth.service: login()

```
constructor(private http: HttpClient) {}

login(username: string, password: string): Observable<any> {
  return this.http.post(
    AUTH_API + 'signin',
    {
      username,
      password,
    },
    httpOptions
  );
}
```

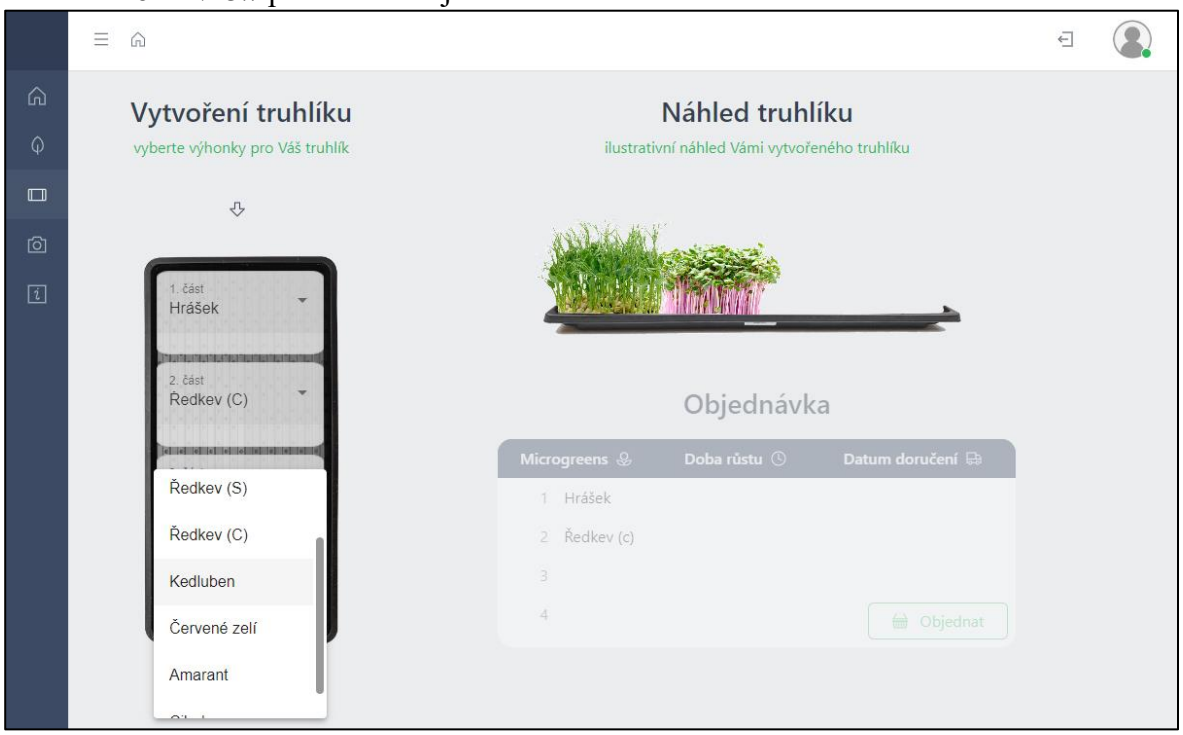
Zdroj: vlastní zpracování

Hesla uživatelů jsou v databázi z bezpečnostních důvodů uložena v šifrované podobě, k čemuž byla využita knihovna *BCryptjs v2.4.3*. Jedná se o nástroj umožňující hashování hesel s použitím algoritmu BCrypt. Proto je heslo zadané uživatelem při přihlašování nejdříve zašifrováno a následně porovnáno se záznamem v databázi. Pokud dojde v serverové části aplikace po přijetí HTTP požadavku k nalezení uživatelského jména a úspěšnému ověření hesla, uživatelské části aplikace je navrácena informace o úspěšném přihlášení uživatele spolu s uživatelskými daty, jak ukazuje dříve popisovaný obrázek 38 část (b). Zpětná vazba je zachycena funkcí *onSubmit()*. V případě úspěšného přihlášení jsou pomocí služby *storage.service* identifikována přidělená oprávnění uživatele, která se nacházejí v uložišti prohlížeče. Uživatel je následně přesměrován na domovskou stránku aplikace, která se aktualizuje a zobrazí uživateli obsah příslušný jeho oprávněním.

4.3.2 Tvorba objednávek

Vytvořená aplikace umožňující autentizaci a autorizaci uživatelů byla rozšířena o funkcionalitu umožňující zákazníkům vytvářet objednávky. Layout pro vytvoření objednávky se skládá ze tří prvků: prvek pro sestavení truhlíku, vizualizace vytvářeného truhlíku a souhrn objednávky. Prvek pro sestavování truhlíku umožňuje uživateli určit výhonky, které má truhlík obsahovat. Vizualizační prvek poskytuje náhled vytvářeného truhlíku a souhrn objednávky slouží ke kontrole parametrů a odeslání objednávky. View pro vytváření objednávky je zachyceno na obrázku 40.

Obrázek 40 View pro tvorbu objednávek



Zdroj: vlastní zpracování

Prvek pro sestavování truhlíku umožňuje uživateli určit složení truhlíku pomocí čtyř select elementů, přičemž každý z nich reprezentuje jednu část sestavovaného truhlíku. Uživateli se po kliknutí na příslušnou část truhlíku rozbalí nabídka s výhonky, které je možné do zvolené části truhlíku vysít. Obsah rozbalovací nabídky výhonků byl vytvořen využitím strukturální direktivy **ngFor* a hodnota zvolené možnosti je obousměrně svázána s proměnnou ve třídě komponentu. Po zvolení druhu výhonku se aktualizuje náhled vytvářeného truhlíku. Select element obsahuje událost *(selectionChange)*, která nastává při každém vybrání výhonku z rozbalovací nabídky. Událost *(selectionChange)* volá funkci *onMgChange()* s parametrem nově zvolené hodnoty výhonku, která na základě parametru aktualizuje příslušnou část vizualizace změnou obrázku výhonků nově zvoleného druhu. Template prvku pro sestavení truhlíku ukazuje ukázka kódu 37. Vizualizace sestavovaného truhlíku slouží především k lepší UX. Jeho cílem je učinit pro zákazníka proces sestavování truhlíku zábavným a zároveň uživateli nastínit, jak bude sestavený truhlík vypadat.

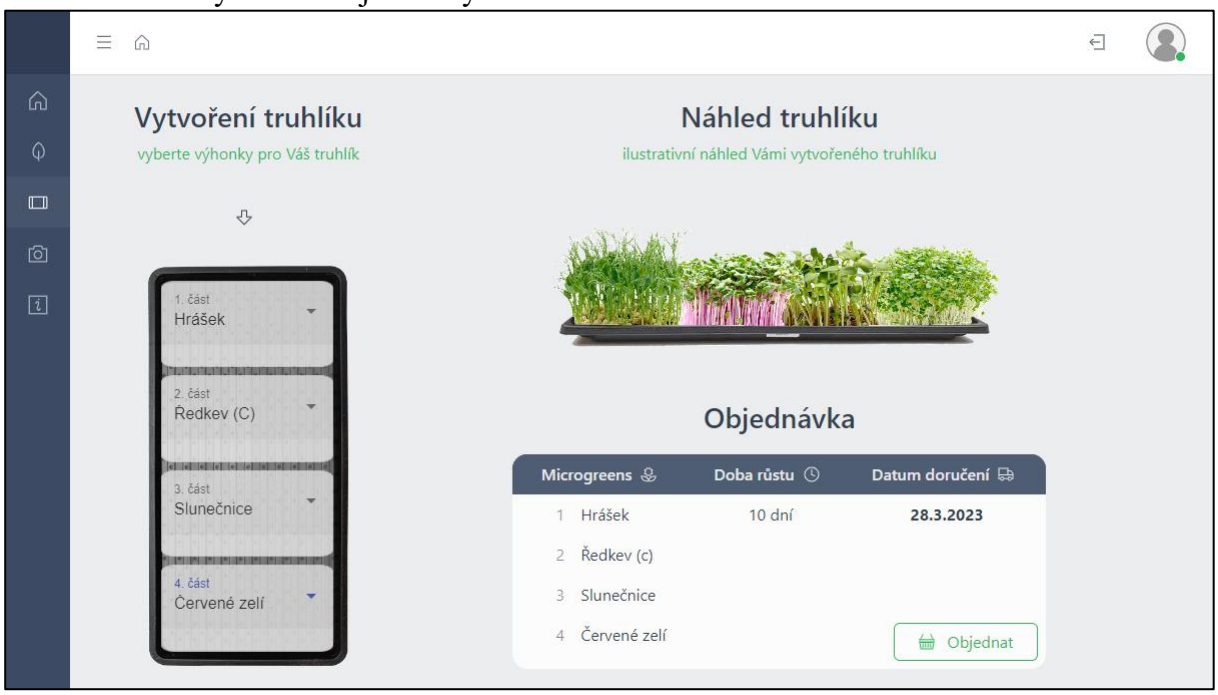
Ukázka kódu 37 Template pro sestavení truhlíku

```
<!-- VERTICAL BOX -->
<div class="boxVerticalDisplay">
  <!-- VERTICAL BOX HEADER -->
  <div class="selectHeader">
    <h3 id="selectHeader">Vytvoření truhlíku</h3>
    <p id="selectHeader" style="color: #2eb85c;">vyberte výhonky pro Váš truhlík</p>
    <div class="arrowContainer">
      <svg cIcon class="animatedIcon me-2" name="cilArrowThickBottom"></svg>
    </div>
  </div>
  <!-- VERTICAL BOX SELECT -->
  <div class="vBoxDiv">
    <div class="vBoxContent">
      <div class="boxPart marginBoxPart d-flex" >
        <mat-form-field appearance="fill" class="boxField" id="firstPart">
          <mat-label>1. část</mat-label>
          <mat-select class="mgSelect" [(value)]="boxPart1"
          (selectionChange)="onMg1Change($event)">
            <mat-option *ngFor="let mg of microgreens" [value]="mg.value">
              {{mg.viewValue}}
            </mat-option>
          </mat-select>
        </mat-form-field>
      </div>
      <div class="boxPart marginBoxPart d-flex">
        <mat-form-field appearance="fill" class="boxField">
          <mat-label>2. část</mat-label>
          …
        </mat-form-field>
      </div>
      <div class="boxPart marginBoxPart d-flex">
        <mat-form-field appearance="fill" class="boxField">
          <mat-label>3. část</mat-label>
          …
        </mat-form-field>
      </div>
      <div class="boxPart d-flex">
        <mat-form-field appearance="fill" class="boxField">
          <mat-label>4. část</mat-label>
          …
        </mat-form-field>
      </div>
    </div>
  </div>
</div>
```

Zdroj: vlastní zpracování

Jakmile uživatel zvolí pro každou část truhlíku druh výhonku, zpřístupní se uživateli souhrn objednávky, kterou může potvrzením formuláře vytvořit. Zpřístupnění sekce pro potvrzení objednávky je zachyceno na obrázku 41. Při potvrzení objednávky je zavolána funkce *onClick()* nacházející se ve třídě komponentu. Funkce připraví potřebná data k odeslání do serverové části aplikace a pomocí služby *user.service* realizuje příslušné API volání. Funkce *onClick()* je zachycena v ukázce kódu 38, realizace požadavku servisní funkcí *user.service* je zachycena v ukázce kódu 39.

Obrázek 41 Vytvoření objednávky



Zdroj: vlastní zpracování

Ukázka kódu 38 order.component: onClick()

```
onClick(event: any) {
    if (this.boxPart1 && this.boxPart2 && this.boxPart3 && this.boxPart4) {
      let username = this.currentUser.username;
      let microgreens = [{
        "MG1":this.boxPart1,
        "MG2":this.boxPart2,
        "MG3":this.boxPart3,
        "MG4":this.boxPart4
      }]
      this.userService.newOrder(username,microgreens).subscribe({
        next: res => {
          let orderDlgRef = this.dialog.open(
            OrderDialogComponent,
            {
              height: '55%',
              width: '40%',
              data: {
                boxTime: this.growTime,
                deliveryDate: this.deliveryDate,
                O_Id: res.O_Id
              }
            },
          )
        },
        error: err => {
          console.log(err);
        }
      });
    }
}
```

Zdroj: vlastní zpracování

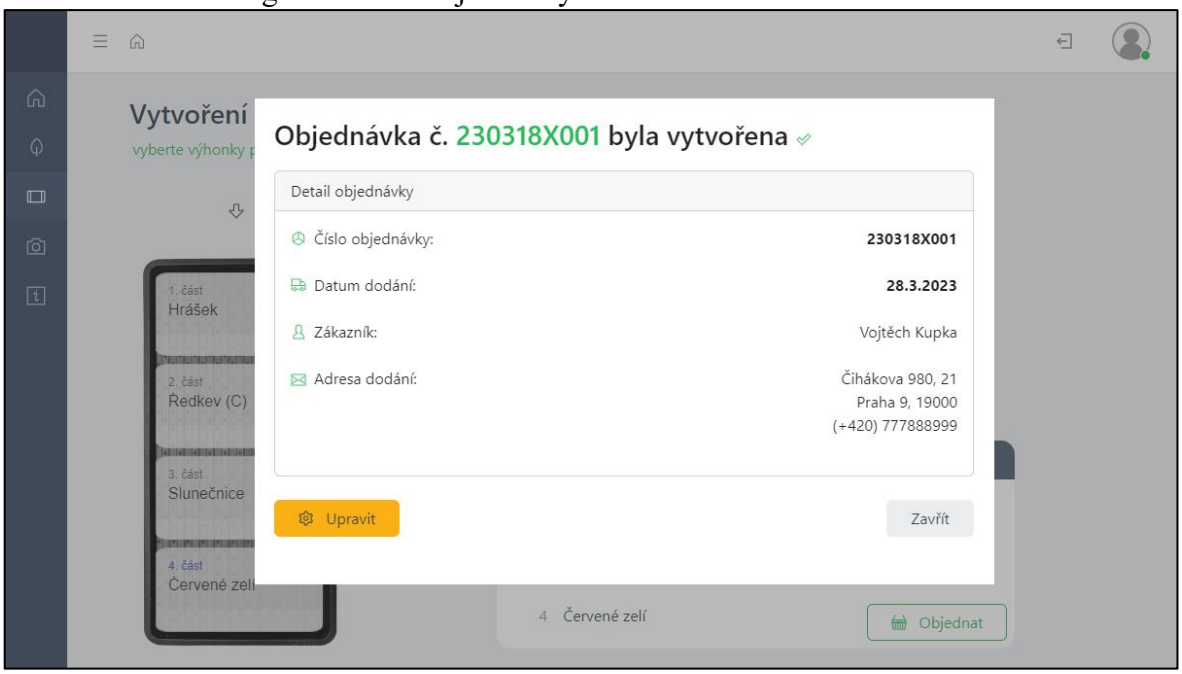
Ukázka kódu 39 user.service

```
export class UserService {
  constructor(private http: HttpClient) {}

  newOrder(username:string, microgreens:any): Observable<any> {
    return this.http.post(API_URL + 'newOrder',
    { username, microgreens },  httpOptions);
  }
}
```

Zdroj: vlastní zpracování

API volání je dále zpracováno příslušnou funkcí v serverové části aplikace. Během zpracování dojde k vygenerování ID objednávky a přidružených truhlíků a následně je nová objednávka spolu s truhlíky zapsána do databáze. Novým truhlíkům je nastaven status *waiting*, jak bylo navrženo v rámci stavového diagramu na obrázku 28 v kapitole 4.2.1 (Modelování systému pomocí UML). V rámci generování úkolů pro zakládání nových truhlíků bude tudíž na jeho základě vygenerován úkol pro obsluhu pěstírny. Po úspěšném zpracování příchozí objednávky jsou do klientské části aplikace navráceny informace o vygenerovaných ID objednávky a truhlíků, které jsou uživateli zobrazeny pomocí dialogu. Dialog obsahuje souhrn informací o přijaté objednávce. Uživatel může zkontrolovat své údaje a v případě potřeby aktualizovat adresu pro doručení objednávky. Dialog je vyobrazen na obrázku 42.

Obrázek 42 Dialog s detailem objednávky

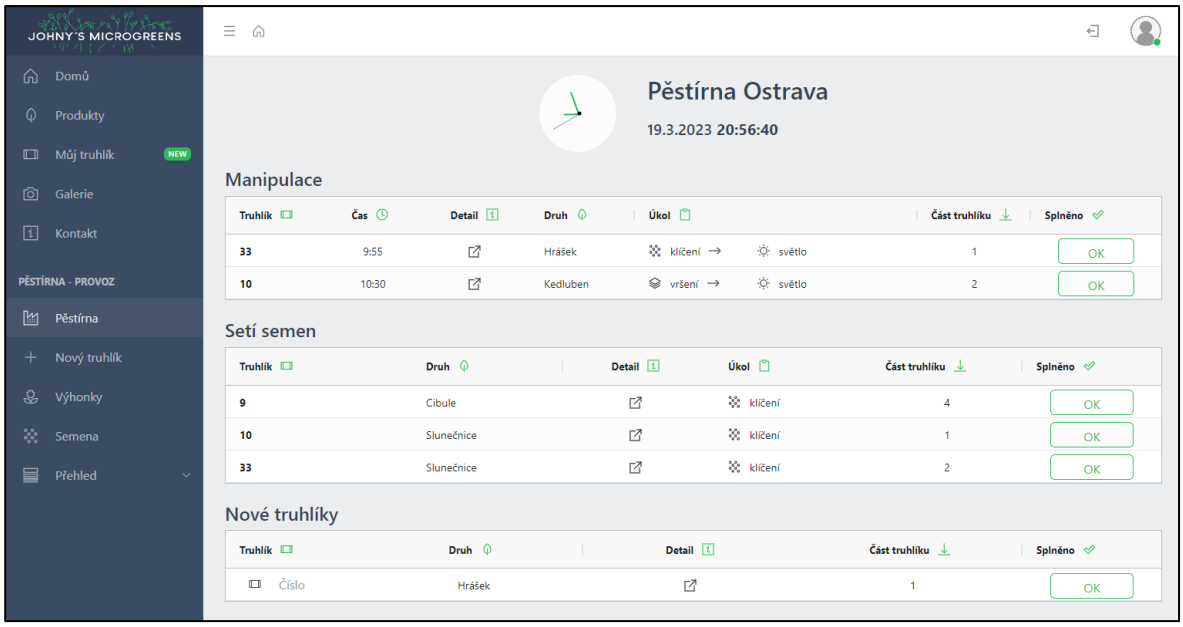


Zdroj: vlastní zpracování

4.3.3 Řízení chodu pěstírny

Aplikace byla dále rozšířena o část *Pěstírna* umožňující řízení chodu pěstírny, přístupnou pouze pro obsluhu pěstírny. Obsluha pěstírny je z pohledu aplikace uživatel disponující oprávněním *moderator*. Po přihlášení takového uživatele je v hlavní navigaci v levé části obrazovky zobrazena navíc sekce *Pěstírna*, která umožňuje obsluze plnit úkoly spojené s provozem pěstírny. View pěstírny je složeno ze tří multifunkčních interaktivních tabulek, přičemž každá z nich zobrazuje seznam úkolů určitého typu, jak bylo popsáno v kapitole 4.2.2 (Modelování procesů pomocí BPMN). View pěstírny je zachyceno na obrázku 43.

Obrázek 43 Sekce Pěstírna



Zdroj: vlastní zpracování

Interaktivní tabulky byly vytvořeny pomocí knihovny AG Grid. Jedná se o pokročilou tabulkovou knihovnu pro webové aplikace, která poskytuje různé možnosti pro zobrazení, filtrování, řazení a editaci dat v tabulkách. Může být integrována s různými datovými zdroji, jako je například RestAPI. Jak je patrné z obrázku 43, tabulka s manipulačními úkoly obsahuje pro každý úkol informaci o čísle truhlíku, optimálním čase splnění úkolu, druhu výhonku, části truhlíku a popisu úkolu. V pravé části tabulky je sloupec s tlačítkem *OK*, který obsluha využije k oznámení splnění daného úkolu. Tabulka obsahuje také sloupec detail, který umožňuje obsluze zobrazení detailu úkolu po kliknutí na ikonku v řádku úkolu. Detail úkolu se uživateli zobrazí pomocí dialogu, který lze pozorovat na obrázku 44. Proces generování a plnění manipulačních úkolů je popsán v kapitole 4.2.2

(Modelování procesů pomocí BPMN) a modelován na Obrázcích 34 a 35. Do sloupců tabulek lze vykreslovat vlastní obsah, jako jsou tlačítka nebo ikony. To je možné díky komponentu pro renderování vlastního obsahu. Ukázka kódu 40 zachycuje definici prvního sloupce tabulky s manipulačními úkoly, kde je uvedeno číslo truhlíku, kterého se úkol týká. V hlavičce sloupce je pomocí vykreslovací komponenty zobrazen nadpis a ikona, které jsou v definici hlavičky sloupce předány vykreslovacímu komponentu jako parametry. Samotný vykreslovací komponent je zachycen v ukázce kódu 41.

Ukázka kódu 40 AG Grid: definice sloupce

```
//Tasks Grid definition
this.manipulationTasksGridOptions = {
  columnDefs: [
    {
      headerName: 'Truhlík',
      headerComponent:IconRendererComponent,
      headerComponentParams: {
        text1:"Truhlík",
        icon2:'cil-mobile-landscape',
      },
      field: 'boxNo',
      width: 110,
      minWidth: 60,
      cellStyle: {
        'font-weight': 'bold',
      }
    },
  ]
}
```

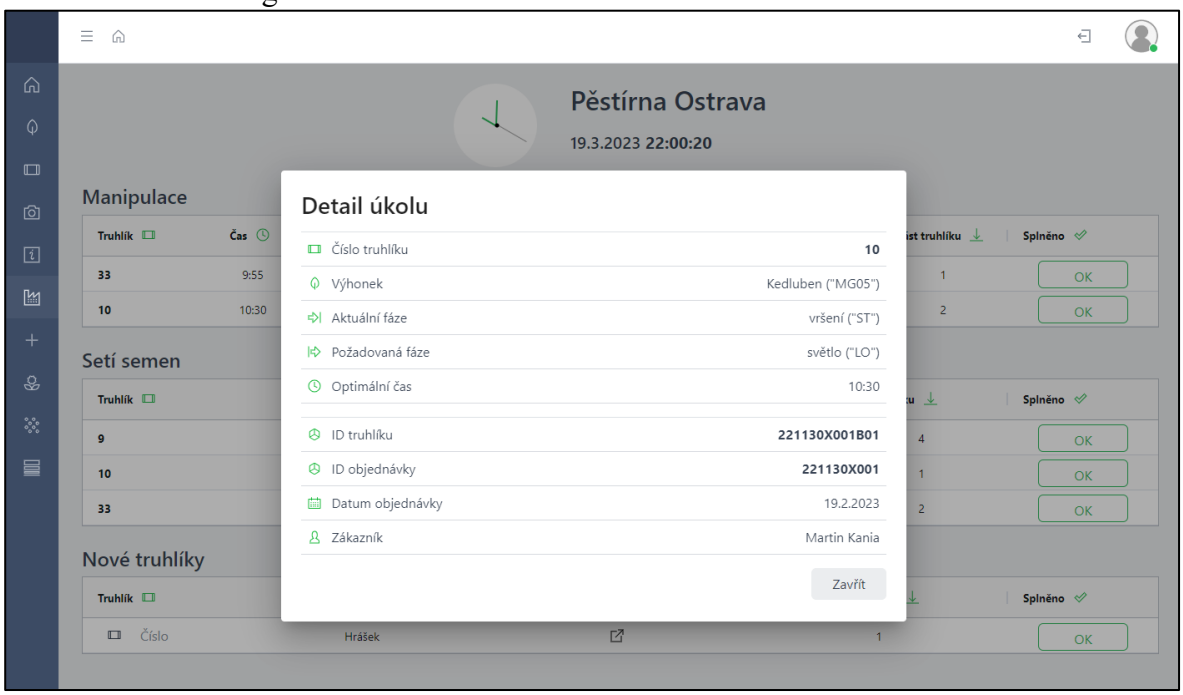
Zdroj: vlastní zpracování

Ukázka kódu 41 Komponenta pro renderování hlavičky sloupce

```
import { Component } from '@angular/core';
import { ICellRendererAngularComp } from 'ag-grid-angular';

@Component({
  selector: 'app-icon-renderer',
  template: '<span id="headerTextR" class="headerTextR">{{text1}}</span>   <svg cIcon
class="me-2 filter-green" name="{{this.icon2}}"></svg>',
  styleUrls: ['./icon-renderer.component.scss']
})

export class IconRendererComponent implements ICellRendererAngularComp {
  text1: string = "Default";
  icon2: any = "";
  agInit(params: any): void {
    this.text1 = params.text1;
    this.icon2 = params.icon2;
  }
  refresh(params: any): boolean {
    return false;
  }
}
```

Zdroj: vlastní zpracování

Obrázek 44 Dialog s detailem úkolu



Zdroj: vlastní zpracování

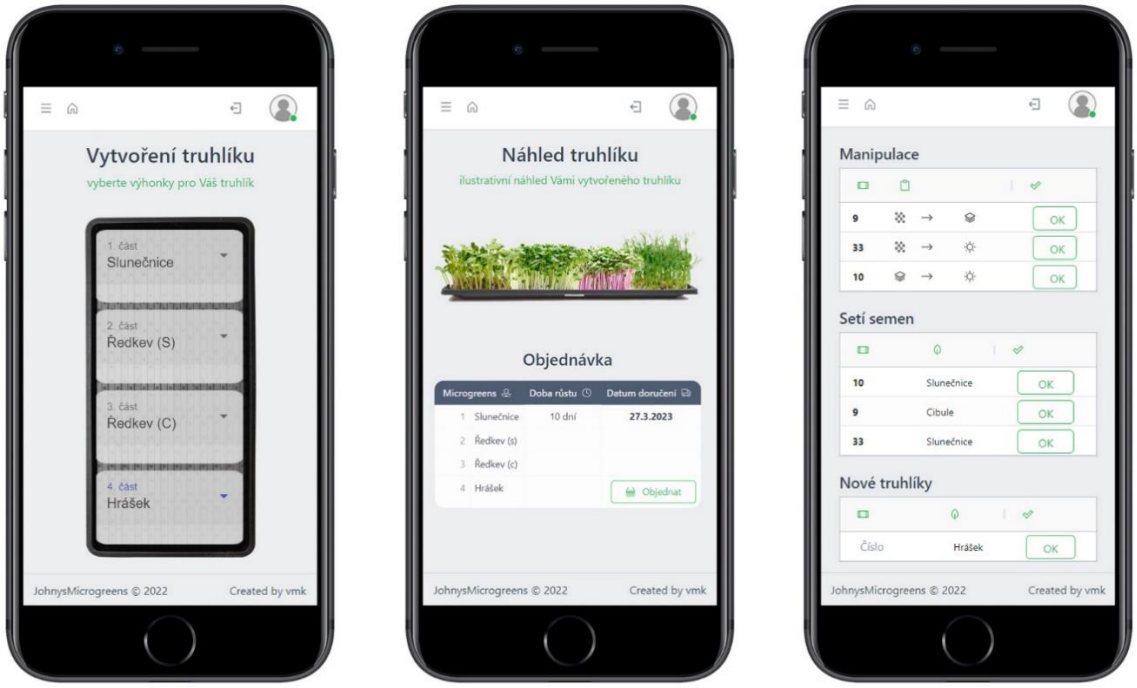
Na obrázku 43 lze dále pozorovat tabulku s úkoly pro výsev semen do již založených truhlíků. Tabulka obsahuje pro každý úkol informaci o čísle truhlíku, části truhlíku a druhu výhonku a úkolu spolu s možností detailního zobrazení úkolu. Procesy generování úkolů pro výsev semen do založených truhlíků a jejich plnění jsou popsány v kapitole 4.2.2 (Modelování procesů pomocí BPMN) a modelovány pomocí BPMN diagramů na Obrázcích 32 a 33.

Tabulka úkolů ve spodní části sekce *Pěstírna* zobrazuje úkoly pro zakládání nových truhlíků na základě příchozích objednávek. Zdrojový kód funkce realizující generování těchto úkolů v serverové části aplikace je vyobrazen v ukázce kódu 1 v příloze B. Ve zdrojovém kódu lze pozorovat získání dat týkajících se všech pěstovaných výhonků spolu s načtením truhlíků, které čekají na založení. Následně dojde k identifikaci nejdéle rostoucích výhonků v každém nalezeném truhlíku. Pro každý truhlík je vytvořen úkol obsahující informaci o ID truhlíku, celkové době potřebné k vyprodukování truhlíku a výhoncích, které bude potřeba do truhlíku vysít jako první při založení truhlíku. Pro splnění úkolu musí obsluha pěstírny nejdříve zadat číslo reálného truhlíku, který k vysetí semen prvních výhonků použije. Poté může potvrdit splnění úkolu. Zdrojový kód funkce serverové části aplikace zpracovávající splnění tohoto typu úkolu se nachází v příloze C. Ukázka kódu 1 v příloze C zachycuje získání dat o výhoncích a truhlících čekajících na založení a aplikační logiku pro identifikaci výhonků, které byly v rámci splnění úkolů zasety.

Ukázka kódu 2 v příloze C zachycuje aktualizaci stavů zasetých výhonků a ukázka kódu 3 v příloze C zachycuje identifikaci zbylých výhonků v rámci truhlíku, u kterých je zapotřebí naplánovat čas výsevu. Ukázka kódu 4 v příloze C zachycuje naplánování výsevu zbylých výhonků do truhlíku a aktualizaci těchto truhlíků v databázi.

Z důvodu dodržení doporučeného rozsahu této práce nebylo možné detailně ukázat všechny funkcionality a procesy vytvořené aplikace. Kromě popsaných částí obsahuje aplikace dále modul pro správu uživatelského profilu, ve kterém může uživatel upravovat své kontaktní a přihlašovací údaje. Součástí aplikace je také sekce galerie prezentující prostory pěstírny a pěstované výhonky spolu s informacemi o výhoncích týkajících se kladného přínosu konzumace výhonků pro zdraví člověka. Uživatel v aplikaci najde také kontakt na obsluhu pěstírny, adresu pěstírny a další potřebné informace. Jedná se vesměs o statické části webové aplikace, které mají primárně informativní charakter. Ústřední částí aplikace je domovská stránka. Zákazník může všechny úkony spojené s používáním aplikace provádět v příslušných sekcích, do kterých se dostane pomocí hlavní navigace, nebo přímo v prostředí domovské stránky. Do View domovské stránky jsou vloženy všechny klíčové komponenty a uživatel se může z domovské stránky přihlásit, sestavit vlastní truhlík i vytvořit objednávku. To může být užitečné především pro uživatele, kteří přistupují k webové aplikaci prostřednictvím mobilních zařízení. Klíčovou vlastností webové aplikace je responzivita jejího zobrazování. Responzivní aplikace umožňuje uživatelům aplikaci používat na různých typech zařízení, jako jsou chytré telefony nebo tablety. Náhled vytvořené aplikace zobrazené na mobilním telefonu je zachycen na obrázku 45. V sekcích (a) a (b) obrázku 45 je mobilní zobrazení části aplikace pro vytváření objednávky a v části (c) obrázku 45 je zachyceno mobilní zobrazení sekce *Pěstírna*. Úkoly v sekci Pěstírna jsou na mobilních zařízeních zobrazovány v minimalistické podobě. Předpokládá se, že obsluha pěstírny bude pro přístup k aplikaci využívat zařízení s širším displejem, aby bylo docíleno zobrazení většího detailu jednotlivých úkolů. Ideálním typem zařízením pro obsluhu pěstírny z hlediska zobrazení a snadné manipulace se jeví tablet.

Obrázek 45 Mobilní zobrazení aplikace



(a) (b) (c)

Zdroj: vlastní zpracování

5 Výsledky a diskuse

V rámci vlastní práce byla provedena analýza provozu konkrétní pěstírny, na jejímž základě byly definovány požadavky na systém pro řízení provozu v pěstírně. Definované procesy probíhající v pěstírně byly brány jako referenční pro návrh a následný vývoj aplikace, tato práce se tudíž nezabývala optimalizací samotného výrobního procesu probíhajícího v pěstírně. Práce se zabývala návrhem a realizací řídicího systému pro podporu definovaného výrobního procesu a jeho optimalizaci z hlediska času a kvality. Následný návrh aplikace a modelování procesů systému na různých úrovních abstrakce napomohli k odhalení některých možných technických problémů, především v oblasti návrhu struktury databáze. Tyto aspekty systému byly v rámci návrhu definovány a diskutovány. Na základě návrhu byla vytvořena webová aplikace umožňující zákazníkům vytvářet objednávky, na jejichž základě systém aplikace plánuje výrobu a instruuje obsluhu pěstírny úkoly pro efektivní chod pěstírny. Jedná se tedy o aplikaci představující typ chytrého e-shopu, který implementuje systém pro řízení chodu pěstírny na základě přijatých objednávek.

5.1 Návrh systému

K návrhu a modelování systému byly využity standardizované jazyky UML a BPMN. Pomocí jazyka UML byl vytvořen UseCase diagram specifikující chování aplikace z pohledu uživatelů a popisující způsoby, jakými bude aplikace používána. Základní objekty aplikace, jejich struktura a vzájemné vazby byly navrženy pomocí diagramu tříd, na jehož základě byla navržena struktura relační databáze. Pomocí diagramu aktivit byly modelovány procesy registrace a přihlašování uživatelů a proces vytvoření objednávky uživatelem. Pomocí sekvenčního diagramu byla znázorněna časová posloupnost aktivit procesu vytvoření objednávky a uzly, které příslušné aktivity zpracovávají. Životní cyklus objednávky a truhlíku byl navržen pomocí stavového diagramu, na kterém lze pozorovat stavy, ve kterých se mohou objekty nacházet, jejich posloupnost a akce způsobující změny těchto stavů. Fyzické rozložení částí aplikace bylo navrženo pomocí diagramu nasazení.

Klíčové procesy realizující generování úkolů a jejich plnění byly vymodelovány pomocí BPMN. Generování úkolů a jejich plnění bylo modelováno formou diagramů spolupráce, umožňujících znázornění interakce mezi klientskou a serverovou částí aplikace. Takto byl vymodelován proces generování manipulačních úkolů, úkolů pro výsev semen

a úkolů pro zakládání nových truhlíků při příchodu nové objednávky. Obdobným způsobem byly vymodelovány reakce systému na splnění daných typů úkolů.

5.2 Vytvořená aplikace

Webová aplikace implementující navržené řešení byla realizována pomocí frameworku Angular za použití programovacího jazyka TypeScipt. Serverové prostředí aplikace bylo realizováno pomocí frameworku Node.js a Express v programovacím jazyce JavaScript. Navržená relační databáze byla realizována pomocí databázového systému MySQL. Jedním ze soudobých trendů při vývoji webových aplikací je používání NoSQL databází, jako je například databázový systém MongoDB umožňující uchovávat soubory v databázi ve formátu JSON. Pokud by byl v případě vytvořené aplikace použit databázový systém MongoDB namísto MySQL, bylo by dosaženo tzv. MEAN vývoje (MongoDB + Express + Angular + Node.js). Využití těchto čtyř open-source technologií pro tvorbu webových aplikací je aktuálně vývojáři často používáno a jedná se o zajímavou alternativu pro tvorbu webových aplikací. Toto alternativní řešení by se lišilo mimo jiné využitím ORM Sequelize.

Při vývoji aplikace byl kladen důraz na používání aktuálních technologií, postupů a návrhových vzorů. Aplikace byla vyvíjena metodou mobile-first, při které byla vytvářená View nejprve optimalizována pro zobrazení na mobilních zařízeních a následně pro zařízení disponujícími širšími typy obrazovek. Komunikace v rámci architektury RestAPI byla navržena tak, aby docházelo k přenosu pouze nezbytných dat pro fungování systému. Důraz byl kladen také na modularitu vytvořeného systému, což zvyšuje jeho přehlednost a zjednodušuje další budoucí vývoj aplikace.

5.2.1 Další vývoj aplikace

Aby mohla být vytvořená aplikace v budoucnosti uvedena do reálného provozu, bude zapotřebí aplikaci rozšířit o některé funkcionality nad rámec této práce. Z pohledu zákazníka by se jednalo zejména o funkcionalitu košíku, umožňující uživatelům platbu kartou po internetu nebo možnost předplatného, tedy periodické realizace objednávky ve zvoleném časovém intervalu. Z pohledu pěstírny aplikaci schází především propojení s účetním systémem, který by byl schopný například generovat faktury a umožňoval by pěstírně spravovat finanční a účetní problematiku. Aplikaci by také bylo dobré rozšířit o sekci umožňující měnit parametry růstových procesů jednotlivých výhonků, jako jsou

délky jednotlivých růstových fází nebo jejich pořadí. Tyto zmíněné a některé další funkcionality jsou otázkou dalšího možného vývoje vytvořené aplikace.

5.2.2 Cena vývoje aplikace

Realizované řešení nebylo vyvíjeno komerčně, a tudíž nelze přesně určit jeho prodejní cenu. Pokud by se ale pěstírna rozhodla nechat si podobné řešení vyvinout na zakázku, znamenalo by to pro ni určitou finanční investici. Je proto žádoucí pokusit se odhadnout cenu vývoje vytvořeného systému. Uvažujeme-li hodinovou sazbu firmy zabývající se vývojem aplikací 1 000,- Kč a odhadneme, že návrh a vývoj aplikace trval 40 pracovních dní, lze základní cenu vývoje aplikace odhadnout na 320 000,- Kč. Otázkou zůstává, zda by byla taková investice pro začínající pěstírnu únosná. Vzhledem k modularitě aplikace by bylo možné systém přizpůsobit i dalším podobným pěstírnám, přesto že by to znamenalo zásah do některých klíčových procesů a významnou úpravu celé aplikace. V takovém případě by se pak dalo uvažovat o prodeji licence k užívání aplikace a náklady na vývoj by tak mohlo nést více pěstíren.

6 Závěr

Tato práce se věnovala návrhu a vývoji systému pro řízení provozu vertikální pěstírny výhonků na základě objednávek od zákazníků. Systém byl navržen na základě analýzy provozu reálné pěstírny a implementován formou webové aplikace. Vytvořená aplikace by mohla sloužit jako nástroj pro řízení produkce v pěstírně a zároveň přispět k optimalizaci výrobního procesu z hlediska kvality a času. Řízení provozu pěstírny na základě příchozích objednávek by zároveň pěstírně umožnilo realizovat štíhlou výrobu.

V teoretické části práce byla zpracována problematika single-page webových aplikací se zaměřením na vývoj webových aplikací pomocí frameworku Angular. Byla zde rozebírána architektura Angular aplikací a vliv komponentů na modularitu systému. Pozornost byla věnována také problematice datových vazeb v rámci komponentu a možnostech sdílení dat mezi jednotlivými komponenty. Dále byla popsána architektura RestAPI a technologie umožňující její implementaci v serverové části aplikace.

Praktická část práce se věnovala analýze požadavků pěstírny, na jejímž základě byla navržena aplikace implementující systém pro řízení produkce v pěstírně. Pro návrh aplikace byl použit jazyk UML, pomocí kterého byla vytvořena detailní specifikace a vizualizace celého systému. V rámci návrhu byly dále modelovány klíčové procesy aplikace formou diagramů vytvořených pomocí standardu BPMN. Vytvořený návrh aplikace napomohl k lepšímu porozumění funkcionalitám systému a odhalil některá jeho problematická místa.

Navržený systém byl implementován formou webové aplikace. Klientská část aplikace byla vytvořena pomocí frameworku Angular, serverová část aplikace byla realizována v prostředí Node a Express. Pro uchovávání dat potřebných k provozu aplikace byl použit relační databázový systém MySQL. UI aplikace byl vyvíjen metodou mobile-first, při vývoji byl kladen důraz na responzivitu zobrazení a UX, aby mohli uživatelé aplikaci pohodlně používat na různých typech zařízení.

7 Seznam použitých zdrojů

[1] LEVINSON, Martin H. The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution: Review of General Semantics. *Simon & Schuster*. Concord: Institute of General Semantics, 2018, **75**(1), 221-222. ISSN 0014-164X. Dostupné také z: https://go.exlibris.link/RnKx5RKv

[2] HOLDENER, Anthony. *Ajax: the definitive guide*. 1. USA: O'Reilly Media, 2008. ISBN 978-0-596-52838-6.

[3] KEMP, Simon. *Digital 2021: Global Overview Report* [online]. Singapore: DataReportal, 2021 [cit. 2023-03-23]. Dostupné z: https://datareportal.com/

[4] CARR, Nicholas. *The Shallows: What the Internet Is Doing to Our Brains*. 1. New York: W. W. Norton & Company, 2020, 320 s. Expanded edition. ISBN 0393357821.

[5] ZHENG, Ting, Marco ARDOLINO, Andrea BACCHETTI a Marco PERONA. The applications of Industry 4.0 technologies in manufacturing context: a systematic literature review. *International journal of production research*. London: Taylor & Francis, 2021, **59**(1), 1922-1954. ISSN 0020-7543. Dostupné z: doi:10.1080/00207543.2020.1824085

[6] XING, YongKang, JiaPeng HUANG a YongYao LAI. Research and Analysis of the Front-end Frameworks and Libraries in E-Business Development. In: *Proceedings of the 2019 11th International Conference on computer and automation engineering*. ACM, 2019, s. 68-72. Dostupné z: doi:10.1145/3313991.3314021

[7] KROTOFF, Tanguy. Front-end frameworks popularity (React, Vue, Angular and Svelte). In: *GitHub* [online]. San Francisco, Kalifornie, USA: GitHub, 2008 [cit. 2022-09-16]. Dostupné z: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190

[8] GREEN, Brad a Shyam SESHADRI. *AngularJS*. 1st. Sebastopol: O'Reilly, 2013, . Dostupné také z: https://go.exlibris.link/8X4fF0Xd

[9] CLI Overview and Command Reference. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-27]. Dostupné z: https://angular.io/cli

[10] KAUFMAN, Nir a Thierry TEMPLIER. *Angular 2 Components*. 1. Birmingham: Packt Publishing, Limited, 2016, . Dostupné také z: https://go.exlibris.link/N2YmVvtg

[11] EL OMARI, Mouad, Mohammed ERRAMDANI a Abdelkader RHOUATI. Getting Model of MVVM Pattern from UML Profile. *International journal of recent contributions from engineering, science & IT*. International Association of Online Engineering (IAOE), 2020, **8**(), 36-47. ISSN 2197-8581. Dostupné z: doi:10.3991/ijes.v8i1.13037

[12] ROBIE, Jonathan a Texcel RESEARCH, ed. What is the Document Object Model?. In: *W3C: Level 1 Document Object Model Specification* [online]. Cambridge, Massachusetts, USA: W3, 1994 [cit. 2022-09-26]. Dostupné z: https://www.w3.org/TR/WD-DOM/introduction.html

[13] JavaScript HTML DOM: The HTML DOM (Document Object Model). In: *W3Schools* [online]. Sandnes, Norsko: Refsnes Data, 1998 [cit. 2022-09-26]. Dostupné z: https://www.w3schools.com/js/js_htmldom.asp

[14] MILLS, Chris a Elliot HAWKINS. Animation performance and frame rate. In: *MDN Web Docs* [online]. San Francisco, Kalifornie, USA: Mozilla Foundation, 2005 [cit. 2022-09-26]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/Performance/Animation_performance_and_frame_rate

[15] AST, Markus. Incremental DOM for Web Components. In: *Studierendensymposium Informatik 2016 der TU Chemnitz*. Chemnitz: Universitätsverlag Chemnitz, 2016, s. 153 - 156. ISBN 978-3944640853.

[16] SAVKIN, Victor. Understanding Angular Ivy: Incremental DOM and Virtual DOM. In: *Nrwl* [online]. Gilbert, Arizona, USA: Narwhal Technologies Inc., 2016 [cit. 2022-09-26]. Dostupné z: https://blog.nrwl.io/understanding-angular-ivy-incremental-dom-and-virtual-dom-243be844bf36

[17] Bootstrap 5 Get Started. In: *W3Schools* [online]. Sandnes, Norsko: Refsnes Data, 1998 [cit. 2022-09-29]. Dostupné z: https://www.w3schools.com/bootstrap5/bootstrap_get_started.php

[18] FAIN, Yakov a Anton MOISEEV. *Angular development with TypeScript*. Second. Shelter Island: Manning, 2019, . Dostupné také z: https://go.exlibris.link/HKR5WXDZ

[19] Angular - Ahead-of-time (AOT) compilation. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-16]. Dostupné z: https://angular.io/guide/aot-compiler

[20] Angular Ivy. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-26]. Dostupné z: https://docs.angular.lat/guide/ivy

[21] Angular - Introduction to Angular concepts. In: *Angular* [online]. 2022 [cit. 2022-09-11]. Dostupné z: https://angular.io/guide/architecture

[22] KUNZ, Gion. *Mastering Angular Components: Build Component-Based User Interfaces Using Angular, 2nd Edition: Build Component-Based User Interfaces Using Angular, 2nd Edition*. Birmingham: Packt Publishing, Limited, 2018, . Dostupné také z: https://go.exlibris.link/N5mfgSq1

[23] LAU, Kung-Kiu a Faris TAWEEL. Data Encapsulation in Software Components. In: *Component-Based Software Engineering*. 1. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, , s. 1-16. ISBN 0302-9743. ISSN 0302-9743. Dostupné z: doi:10.1007/978-3-540-73551-9_1

[24] LAROCCA, Kim. 5 Benefits Of Component-Based Development: How To Drastically Improve Your Development Process. In: *Medium* [online]. 2012 [cit. 2022-09-18]. Dostupné z: https://medium.com/newyorkpublicradiodigital/5-benefits-of-component-based-development-90af513bb7d2

[25] DAYLEY, Brad, Brendan DAYLEY a Caleb DAYLEY. *Learning Angular*. Second. Indianapolis, IN: Addison-Wesley, 2018, . Dostupné také z: https://go.exlibris.link/l54GWZY3

[26] Introduction to modules. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-28]. Dostupné z: https://angular.io/guide/architecture-modules

[27] NgModules. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-28]. Dostupné z: https://angular.io/guide/ngmodules

[28] Templates and views. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-19]. Dostupné z: https://angular.io/guide/architecture-components

[29] Glossary: view. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-19]. Dostupné z: https://angular.io/guide/glossary#view

[30] View encapsulation. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-26]. Dostupné z: https://angular.io/guide/view-encapsulation

[31] Shadow Dom vs. Virtual Dom: A Web Developer's Guide. In: *Testim company* [online]. Tel Aviv-Yafo, Izrael: testim, 2021 [cit. 2022-09-26]. Dostupné z: https://www.testim.io/blog/shadow-dom-vs-virtual-dom/

[32] Glossary: service. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-21]. Dostupné z: https://angular.io/guide/glossary#service

[33] Dependency injection in Angular. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-21]. Dostupné z: https://angular.io/guide/dependency-injection-overview

[34] Glossary: directive. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-22]. Dostupné z: https://angular.io/guide/glossary#directive

[35] Built-in directives. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-22]. Dostupné z: https://angular.io/guide/built-in-directives

[36] Understanding Pipes. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-27]. Dostupné z: https://angular.io/guide/pipes-overview#built-in-pipes

[37] Transforming Data Using Pipes. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-09-27]. Dostupné z: https://angular.io/guide/pipes#creating-pipes-for-custom-data-transformations

[38] Understanding binding. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/binding-overview

[39] Angular change detection and runtime optimization. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/change-detection

[40] Binding syntax. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/binding-syntax

[41] Property binding. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/property-binding

[42] Displaying values with interpolation. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-12]. Dostupné z: https://angular.io/guide/interpolation

[43] Attribute binding. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/attribute-binding

[44] Class and style binding. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/class-binding

[45] Template statements. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/template-statements

[46] Event binding. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/event-binding

[47] NgModel. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-06]. Dostupné z: https://angular.io/api/forms/NgModel#ngmodel

[48] FARHI, Oren a SpringerLink SLUŽBA). *Reactive Programming with Angular and ngrx: Learn to Harness the Power of Reactive Programming with RxJS and ngrx Extensions*. 1. Berkeley, CA: Apress, 2017, . Dostupné také z: https://go.exlibris.link/2WHvVC6k

[49] The RxJS library. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-04]. Dostupné z: https://angular.io/guide/rx-library

[50] Event-driven architecture style. In: *Microsoft Learn* [online]. Redmond, Washington, USA: Microsoft, 2022 [cit. 2022-10-04]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven

[51] Sharing data between child and parent directives and components. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-08]. Dostupné z: https://angular.io/guide/inputs-outputs#sharing-data-between-child-and-parent-directives-and-components

[52] ViewChild. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-12]. Dostupné z: https://angular.io/api/core/ViewChild#description

[53] Angular @ViewChild: In-Depth Explanation (All Features Covered). In: *Angular University* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-12]. Dostupné z: https://blog.angular-university.io/angular-viewchild/

[54] BehaviorSubject. In: *RxJS: Reactive Extensions Library for JavaScript* [online]. 2022 [cit. 2022-10-12]. Dostupné z: https://rxjs.dev/api/index/class/BehaviorSubject

[55] Express/Node introduction. In: *MDN: Resources for Developers, by Developers* [online]. Mountain View, Kalifornie, USA: Mozilla Corporation, 2022 [cit. 2022-10-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

[56] KRAUSE, Jörg a SpringerLink SLUŽBA). *Programming Web Applications with Node, Express and Pug*. Berkeley, CA: Apress, 2017, . Dostupné také z: https://go.exlibris.link/LJFK7V4g

[57] HAHN, Evan. *Express in Action*. 1. USA: Manning Publications, 2016. ISBN 9781617292422.

[58] HERRON, David. *Node.js Web Development - Third Edition*. Packt Publishing, 2016, . Dostupné také z: https://go.exlibris.link/LNsNSGFm

[59] Communicating with backend services using HTTP. In: *Angular* [online]. Mountain View, Kalifornie, USA: Google, 2022 [cit. 2022-10-23]. Dostupné z: https://angular.io/guide/http

[60] DALIMUNTHE, Syabdan, Joeharsyah REZA a Asep MARZUKI. The Model for Storing Tokens in Local Storage (Cookies) Using JSON Web Token (JWT). *Journal of Applied Engineering and Technological Science*. 2022, **3**(), 149-155. ISSN 2715-6087. Dostupné z: doi:10.37385/jaets.v3i2.662

[61] LAURENČÍK, Marek a Michal BUREŠ. *SQL: podrobný průvodce uživatele: podrobný průvodce uživatele*. První vydání. Praha: Grada Publishing, 2018, . Dostupné také z: https://go.exlibris.link/RSK7CD1L

[62] MIR, Shabir, Mohammad MIR a Manzoor SHAH. Microgreens: Production, shelf life and bioactive components. *Critical Reviews in Food Science and Nutrition*. 2016, **2016**(57), 00-00. Dostupné z: doi:10.1080/10408398.2016.1144557

[63] CHOE, Uyory, Thomas WANG a Liangli YU. The science behind microgreens as an exciting new food for the 21st century. *Journal of agricultural and food chemistry*. ACS Publications, 2018, **66**(44), 11519-11530.

[64] GHOORA, Manjula, Dandamudi BABU a N. SRIVIDYA. Nutrient composition, oxalate content and nutritional ranking of ten culinary microgreens. *Journal of food composition and analysis*. Elsevier Inc, 2020, **91**(), 103495. ISSN 0889-1575. Dostupné z: doi:10.1016/j.jfca.2020.103495

[65] XIAO, Zhenlei, Gene LESTER, Yaguang LUO a Qin WANG. Assessment of Vitamin and Carotenoid Concentrations of Emerging Food Products: Edible Microgreens. *Journal of Agricultural and Food Chemistry* [online]. 2012, **60**(31), 7644-7651 [cit. 2022-09-17]. ISSN 0021-8561. Dostupné z: doi:10.1021/jf300459b

[66] PETTERSEN, Jostein. Defining lean production: some conceptual and practical issues: some conceptual and practical issues. *TQM journal*. Bingley: Emerald Group Publishing Limited, 2009, **21**(), 127-142. Dostupné z: doi:10.1108/17542730910938137

[67] BLAHA, Michael a James RUMBAUGH. *Object-Oriented Modeling and Design with UML*. 2. Indie: Pearson, 2005, 496 s. ISBN 978-0130159205.

[68] ENSTROM, David W. *A Simplified Approach to IT Architecture with BPMN*. 1. Indie: iUniverse, 2016. ISBN 978-1491784976.

8 Seznam obrázků, tabulek, zdrojových kódů a zkratek

8.1 Seznam obrázků

8.2 Seznam tabulek

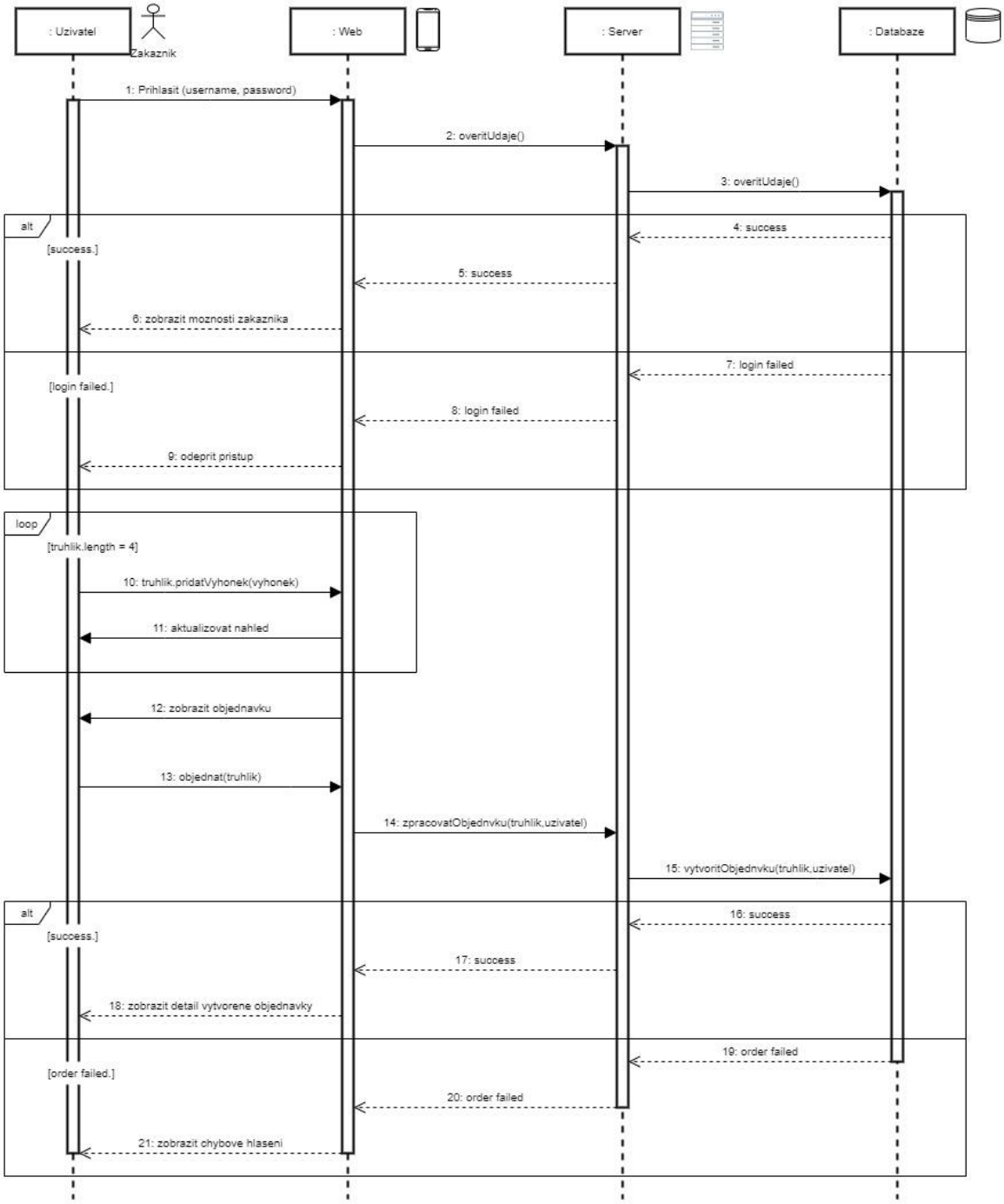
8.3 Seznam ukázek zdrojového kódu

8.4 Seznam použitých zkratek

API	Application Programming Interface
AOT	Ahead Of Time (Compilation)
BPMN	Business Process Model and Notation
BSS	Browser Session Storage
CLI	Command Line Interface
CORS	Cross Origin Resource Sharing
CSS	Cascading Style Sheets
DOM	Document Object Model
EDA	Event Driven Architecture
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDOM	Incremental DOM
JIT	Just In Time (Compilation)
MVC	Model View Controller
MVVM	Model View ViewModel
NGC	Angular Compiler
NPM	Node Package Manager
OOP	Object Oriented Programming
ORM	Object Relational Mapping
RXJS	Reactive Extensions Library for JavaScript
SPWA	Single Page Web Application
SQL	Structure Querz Language
UI	User Interface
UML	Unified Modelling Language
URL	Uniform Resource Locator
UX	User Experience
VDOM	Virtual DOM

Příloha A Sekvenční diagram

Obrázek 1 Přihlášení uživatele a vytvoření objednávky



Příloha B Generování úkolů

Ukázka kódu 1 Úkoly pro zakládání nových truhlíků

```
exports.getNewBoxes = async (req, res) => {
  try {
    const MGs = await Microgreens.findAll({})
    .then(data => {
      if (data) {
        MGsData = data;
      } else {
        res.status(404).send({ message: err.message });
      }
    })
    .catch(err => { res.status(500).send({ message: err.message }); });
  } catch (err) { return res.status(500).send({ message: err.message }); }
  try {
    const boxes = await Boxes.findAll({
      where: {
        status: 'waiting'
      }
    }).then(data => {
      if (data) {
        try {
          const newBoxes = [];
          for (let i = 0; i < data.length; i++) {
            let newBox = { B_Id: "", B_time: 0, MG_Ids: {} };
            for (let j = 0; j < MGsData.length; j++) {
              for (let k = 1; k <= 4; k++) {
                if (data[i].dataValues['MG' + k] == MGsData[j].dataValues.MG_Id) {
                  let MGXTime = MGsData[j].dataValues.F1_time
                    + MGsData[j].dataValues.F2_time
                    + MGsData[j].dataValues.F3_time
                    + MGsData[j].dataValues.F4_time;
                  if (MGXTime > newBox.B_time) {
                    newBox.B_Id = data[i].dataValues.B_Id,
                    newBox.MG_Ids = {};
                    newBox.MG_Ids['mg' + k] = MGsData[j].dataValues.MG_Id;
                    newBox.B_time = MGXTime;
                  } else if (MGXTime == newBox.B_time)
                  {
                    newBox.MG_Ids['mg' + k] = MGsData[j].dataValues.MG_Id;
                  }
                }
              }
            }
            newBoxes.push(newBox)
          }
          res.send(newBoxes);
        }catch (err) { return res.status(404).send({ message: err.message }); }
      } else { res.status(404).send({ message: err.message }); }
    })
    .catch(err => { res.status(500).send({ message: err.message }); });
  } catch (err) { return res.status(500).send({ message: err.message }); }
};
```

Příloha C Plnění úkolů

Ukázka kódu 1 Založení nového truhlíku: Identifikace zasetých výhonků

```
exports.startBox = async (req, res) => {
  let today = new Date();
  try {
    // ZÍSKÁNÍ DAT O VÝHONCÍCH
    …
  } catch (err) { return res.status(500).send({ message: err.message }); }
  try {
    // ZÍSKÁNÍ DAT O TRUHLÍCÍCH
    …
    }).then(data => {
      if (data) {
        try {

          // IDENTIFIKACE ZASETÝCH VÝHONKŮ
          for (let j = 0; j < MGsData.length; j++) {
            for (let k = 1; k <= 4; k++) {
              if (data.dataValues['MG' + k] == MGsData[j].dataValues.MG_Id) {
                let MGXTime = MGsData[j].dataValues.F1_time + MGsData[j].dataValues.F2_time
                + MGsData[j].dataValues.F3_time + MGsData[j].dataValues.F4_time;
                if (MGXTime > Box.B_time) {
                  Box.MGStart_Ids = [];
                  Box.MGStart = [];
                  Box.MGStart_Ids.push(MGsData[j].dataValues.MG_Id);
                  Box.MGStart.push('MG' + k);
                  Box.B_time = MGXTime;
                }
                else if (MGXTime == Box.B_time)
                {
                  Box.MGStart_Ids.push(MGsData[j].dataValues.MG_Id);
                  Box.MGStart.push('MG' + k);
                }
              }
            }
          }


          // ZMĚNA STAVU ZASETÝCH VÝHONKŮ
          // IDENTIFIKACE VÝHONKŮ ČEKAJÍCÍCH NA VÝSEV
          // PLÁNOVÁNÍ VÝSEVU DALŠÍCH VÝHONKŮ

          Boxes.update({ status: "growing" }, { where: { B_Id: Box.B_Id }});
          Boxes.update({ T_No: req.body.B_No }, { where: { B_Id: Box.B_Id }});
          res.send({ "boxId": Box.B_Id, "message": "Nový truhlík byl zasazen." });
        } catch (err) { return res.status(404).send({ message: err.message });
        }
      } else {
        res.status(404).send({ message: err.message });
      }
    })
    .catch(err => {
      res.status(500).send({ message: err.message });
    });
  } catch (err) {
    return res.status(500).send({ message: err.message });
  }
};
```

Ukázka kódu 2 Založení nového truhlíku: Změna stavu zasetých výhonků

```
exports.startBox = async (req, res) => {
  let today = new Date();
  try {
    // ZÍSKÁNÍ DAT O VÝHONCÍCH
  } catch (err) { return res.status(500).send({ message: err.message }); }
  try {
    // ZÍSKÁNÍ DAT O TRUHLÍCÍCH
    }).then(data => {
      if (data) {
        try {

          // IDENTIFIKACE ZASETÝCH VÝHONKŮ

          // ZMĚNA STAVU ZASETÝCH VÝHONKŮ
          for (let i = 0; i < Box.MGStart_Ids.length; i++) {
            for (let j = 0; j < MGsData.length; j++) {
              if (Box.MGStart_Ids[i] == MGsData[j].dataValues.MG_Id) {
                switch(Box.MGStart[i]){
                  case "MG1":
                    Boxes.update({ MG1_F: MGsData[j].dataValues.F1 },
                      { where: { B_Id: Box.B_Id }});
                    Boxes.update({ MG1_FStartTime: today }, { where: { B_Id: Box.B_Id }});
                    break;
                  case "MG2":
                    Boxes.update({ MG2_F: MGsData[j].dataValues.F1 },
                      { where: { B_Id: Box.B_Id }});
                    Boxes.update({ MG2_FStartTime: today }, { where: { B_Id: Box.B_Id }});
                    break;
                  case "MG3":
                    Boxes.update({ MG3_F: MGsData[j].dataValues.F1 },
                      { where: { B_Id: Box.B_Id }});
                    Boxes.update({ MG3_FStartTime: today }, { where: { B_Id: Box.B_Id }});
                    break;
                  case "MG4":
                    Boxes.update({ MG4_F: MGsData[j].dataValues.F1 },
                      { where: { B_Id: Box.B_Id }});
                    Boxes.update({ MG4_FStartTime: today }, { where: { B_Id: Box.B_Id }});
                    break;
                }
              }
            }
          }

          // IDENTIFIKACE VÝHONKŮ ČEKAJÍCÍCH NA VÝSEV
          // PLÁNOVÁNÍ VÝSEVU DALŠÍCH VÝHONKŮ

          Boxes.update({ status: "growing" }, { where: { B_Id: Box.B_Id }});
          Boxes.update({ T_No: req.body.B_No }, { where: { B_Id: Box.B_Id }});
          res.send({ "boxId": Box.B_Id, "message": "Nový truhlík byl zasazen." });
        } catch (err) { return res.status(404).send({ message: err.message });
        }
      } else { res.status(404).send({ message: err.message });
      }
    })
    .catch(err => { res.status(500).send({ message: err.message }); });
  } catch (err) {
    return res.status(500).send({ message: err.message });
  }
};
```

Ukázka kódu 3 Založení nového truhlíku: Identifikace výhonků k naplánování

```
exports.startBox = async (req, res) => {
  let today = new Date();
  try {
    // ZÍSKÁNÍ DAT O VÝHONCÍCH
    …
  } catch (err) { return res.status(500).send({ message: err.message }); }
  try {
    // ZÍSKÁNÍ DAT O TRUHLÍCÍCH
    …
    }).then(data => {
      if (data) {
        try {

          // IDENTIFIKACE ZASETÝCH VÝHONKŮ
          // ZMĚNA STAVU ZASETÝCH VÝHONKŮ

          // IDENTIFIKACE VÝHONKŮ ČEKAJÍCÍCH NA VÝSEV
          for (let i = 0; i < Box.MGStart.length; i++) {
            let index = Box.MGPlan.indexOf(Box.MGStart[i]);
            if (index > -1) {
              Box.MGPlan.splice(index, 1);
            }
          }
          for (let i = 0; i < Box.MGPlan.length; i++) {
            Box.MGPlan_Ids.push(data.dataValues[Box.MGPlan[i]]);
          }


          // PLÁNOVÁNÍ VÝSEVU DALŠÍCH VÝHONKŮ

          Boxes.update({ status: "growing" }, { where: { B_Id: Box.B_Id }});
          Boxes.update({ T_No: req.body.B_No }, { where: { B_Id: Box.B_Id }});
          res.send({ "boxId": Box.B_Id, "message": "Nový truhlík byl zasazen." });
        } catch (err) { return res.status(404).send({ message: err.message });
        }
      } else { res.status(404).send({ message: err.message });
      }
    })
    .catch(err => { res.status(500).send({ message: err.message }); });
  } catch (err) {
    return res.status(500).send({ message: err.message });
  }
};
```

Ukázka kódu 4 Založení nového truhlíku: Plánování výsevu dalších výhonků

```
exports.startBox = async (req, res) => {
  let today = new Date();
  try {
    // ZÍSKÁNÍ DAT O VÝHONCÍCH
    …
  } catch (err) { return res.status(500).send({ message: err.message }); }
  try {
    // ZÍSKÁNÍ DAT O TRUHLÍCÍCH
    …
    }).then(data => {
      if (data) {
        try {

          // IDENTIFIKACE ZASETÝCH VÝHONKŮ
          // ZMĚNA STAVU ZASETÝCH VÝHONKŮ
          // IDENTIFIKACE VÝHONKŮ ČEKAJÍCÍCH NA VÝSEV

          // PLÁNOVÁNÍ VÝSEVU DALŠÍCH VÝHONKŮ
          for (let i = 0; i < Box.MGPlan_Ids.length; i++) {
            for (let j = 0; j < MGsData.length; j++) {
              if (Box.MGPlan_Ids[i] == MGsData[j].dataValues.MG_Id) {
                let MGXTime = MGsData[j].dataValues.F1_time + MGsData[j].dataValues.F2_time
                  + MGsData[j].dataValues.F3_time + MGsData[j].dataValues.F4_time;
                let delay = today.addDays(Box.B_time - MGXTime);
                switch(Box.MGPlan[i]){
                  case "MG1":
                    Boxes.update({ MG1_F: "ZZ" }, { where: { B_Id: Box.B_Id }});
                    Boxes.update({ MG1_FStartTime: delay }, { where: { B_Id: Box.B_Id }});
                    break;
                  case "MG2":
                    Boxes.update({ MG2_F: "ZZ" }, { where: { B_Id: Box.B_Id }});
                    Boxes.update({ MG2_FStartTime: delay }, { where: { B_Id: Box.B_Id }});
                    break;
                  case "MG3":
                    Boxes.update({ MG3_F: "ZZ" }, { where: { B_Id: Box.B_Id }});
                    Boxes.update({ MG3_FStartTime: delay }, { where: { B_Id: Box.B_Id }});
                    break;
                  case "MG4":
                    Boxes.update({ MG4_F: "ZZ" }, { where: { B_Id: Box.B_Id }});
                    Boxes.update({ MG4_FStartTime: delay }, { where: { B_Id: Box.B_Id }});
                    break;
                }
              }
            }
          }


          Boxes.update({ status: "growing" }, { where: { B_Id: Box.B_Id }});
          Boxes.update({ T_No: req.body.B_No }, { where: { B_Id: Box.B_Id }});
          res.send({ "boxId": Box.B_Id, "message": "Nový truhlík byl zasazen." });
        } catch (err) { return res.status(404).send({ message: err.message });
        }
      } else { res.status(404).send({ message: err.message });
      }
    })
    .catch(err => { res.status(500).send({ message: err.message }); });
  } catch (err) {
    return res.status(500).send({ message: err.message });
  }
};
```