Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství



Diplomová práce

Návrh a realizace backendového řešení s grafovou databází Neo4j v prostředí Node.js

Štěpán Lebeda

© 2026 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Štěpán Lebeda

Informatika

Název práce

Návrh a relizace backendového řešení s grafovou databází Neo4j v prostředí Node.js

Název anglicky

Design and Implementation of a Backend Solution with the Neo4j Graph Database in a Node.js Environment

---

Cíle práce

Cílem této diplomové práce je navrhnout a implementovat backendové řešení pro aplikaci, která usnadní rozmístění jednotlivých oddělení firmy do konkrétních pater a budov na základě definovaných kapacitních a organizačních požadavků. Systém umožní nastavit kapacity budov a jejich jednotlivých pater, velikosti oddělení, a zároveň definovat vzájemné vazby mezi odděleními na základě úrovně spolupráce. Součástí řešení bude možnost pevně přiřadit vybraná oddělení k určitým lokacím. Klíčovým cílem je využít grafovou databázi Neo4j pro efektivní modelování a vyhodnocování vztahů mezi jednotlivými entitami a implementovat algoritmus pro optimalizované rozmístění oddělení dle zadaných parametrů. Backendová aplikace bude realizována v prostředí Node.js

Metodika

První část práce bude popisovat přehled nástrojů a technik použitých v druhé části práce. Druhá část práce bude softwarová dokumentace aplikace popsané v cíli práce od fáze popisu požadavků až do pilotního provozu. Součástí práce bude také stručný uživatelský návod a popis použitých algoritmů.

Doporučený rozsah práce

60 – 80 stran

Klíčová slova

Node.js; Neo4j; backend; javascript; alokace oddělení

Doporučené zdroje informací

LYON, William. *Full stack GraphQL applications : with React, Node.js, and Neo4j.* Shelter Island, New York: Manning Publication Co., 2022. ISBN 9781617297038.

ROBINSON, Ian, WEBBER, Jim, and EIFREM, Emil (O'Reilly). Graph Databases. Neo Technology, Inc., 2015, 978-1-491-93089-2

Van BRUGGEN, Rik, Learning Neo4j. Published by Packt Publishing Ltd. Livery Place 35 Livery Street Birmingham B3 2PB, UK. 2025, ISBN 978-1-84951-716-4

Předběžný termín obhajoby

2025/26 LS – PEF

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 26. 02. 2026

doc. Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 06. 03. 2026

prof. Ing. Lukáš Čechura, Ph.D.

Děkan

V Praze dne 30. 03. 2026

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Návrh a realizace backendového řešení s grafovou databází Neo4j v prostředí Node.js" 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. Prohlašuji, že jsem nástroje AI využil v souladu s vnitřními předpisy univerzity a principy akademické integrity a etiky.

V Praze dne 30.3.2026 _____

Poděkování

Rád bych touto cestou poděkoval doc. Ing. Vojtěchu Merunkovi, Ph.D. za vedení mé diplomové práce a za poskytnuté rady při jejím vypracování. Dále bych rád poděkoval své rodině a blízkým, kteří mi byli oporou během celého studia.

Návrh a realizace backendového řešení s grafovou databází Neo4j v prostředí Node.js

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací backendového řešení pro optimalizaci rozmístění oddělení v rámci organizační struktury firmy s využitím grafové databáze. Cílem práce je vytvořit systém, který efektivně přiřazuje jednotlivá oddělení do budov a pater s ohledem na jejich kapacitu a vzájemné vazby, jako je požadavek na blízkost spolupracujících týmů. V teoretické části jsou popsány principy webových aplikací, grafových databází a optimalizačních přístupů vhodných pro řešení daného problému. Dále jsou představeny technologie použité při implementaci, zejména Node.js a Neo4j AuraDB.

Praktická část se zaměřuje na návrh datového modelu, implementaci aplikační logiky a návrh algoritmu pro přiřazování oddělení. Tento algoritmus primárně usiluje o umístění oddělení na stejné patro a v případě nedostatečné kapacity využívá sousední patra. Součástí práce je také návrh REST API, které umožňuje správu entit a spouštění optimalizačního procesu. Funkčnost řešení je ověřena pomocí testovacích scénářů, které demonstrují správnost a efektivitu navrženého přístupu.

Výsledkem práce je funkční backendová aplikace, která přináší flexibilní a škálovatelné řešení problému rozmístění oddělení a zároveň ukazuje výhody grafových databází při práci s relačně propojenými daty.

Klíčová slova: Node.js; Neo4j; grafová databáze; backend; javascript; alokace oddělení; optimalizace; REST API; organizační struktura

Design and Implementation of a Backend Solution with the Neo4j Graph Database in a Node.js Environment

Abstract

This diploma thesis focuses on the design and implementation of a backend solution for optimizing the placement of departments within a company's organizational structure using a graph database. The main objective is to develop a system that efficiently assigns departments to buildings and floors while considering capacity constraints and relationships between departments, such as the need for proximity between collaborating teams. The theoretical part describes the principles of web applications, graph databases, and optimization approaches suitable for solving this problem. It also introduces the technologies used in the implementation, particularly Node.js and Neo4j AuraDB.

The practical part focuses on the design of the data model, implementation of the application logic, and development of an assignment algorithm. This algorithm primarily aims to place departments on the same floor and, if capacity constraints prevent this, utilizes adjacent floors. The solution also includes the design of a REST API for managing entities and triggering the optimization process. The functionality of the system is verified through testing scenarios that demonstrate the correctness and efficiency of the proposed approach.

The result of this thesis is a functional backend application that provides a flexible and scalable solution for department placement while highlighting the advantages of graph databases when working with highly interconnected data.

Keywords: Node.js; Neo4j; graph database; backend; javascript; allocation of departments; optimization; REST API; organizational structure

Obsah

1 Úvod **12**

2 Cíl práce a metodika **13**

2.1 Cíl práce 13

2.2 Metodika 13

3 Teoretická část práce **14**

3.1 Informační systémy pro podporu rozhodování 14

3.2 Modelování dat v informačních systémech 15

3.2.1 Relační datový model 16

3.2.2 Grafový datový model 17

3.3 Grafové databáze 18

3.3.1 Oblasti využití 18

3.3.2 Výhody oproti relačním databázím 19

3.3.3 Neo4j 20

3.4 Architektura webových aplikací 20

3.4.1 Klient – server architektura 21

3.4.2 REST architektura 22

3.5 Backendové technologie 23

3.5.1 JavaScript 24

3.5.2 Node.js 25

3.5.3 Express 26

3.6 Optimalizační úlohy a lineární programování 27

3.6.1 Optimalizační úlohy s omezeními 27

3.6.2 Kombinatorická optimalizace 28

3.6.3 Lineární programování 29

3.6.4 Celočíselné lineární programování 30

3.7 Modelování softwarových systémů 31

3.7.1 UML 31

3.7.2 Diagram případů užití 32

3.7.3 Diagram tříd 33

3.8 Testování webových API 34

3.8.1 Manuální testování API 35

3.8.2 Automatizované testování API 36

3.8.3 Testovací framework Jest 37

3.8.4 Knihovna Supertest 38

3.9 Verzovací systémy 39

3.9.1 Git 39

3.9.2 GitHub 40

4 Vlastní práce 41
4.1 Analýza zadání 41
4.1.1 Vznik požadavku a návrh řešení 41
4.1.2 Funkční požadavky systému 42
4.1.3 Nefunkční požadavky 44
4.2 Návrh systému 45
4.2.1 Datový model systému 45
4.2.2 Diagram případů užití 48
4.3 Implementace systému 50
4.3.1 Architektura backendové aplikace 50
4.3.2 Komunikace s databází Neo4j 52
4.3.3 Implementace REST API 52
4.3.4 Implementace práce s mapou rozmístění 54
4.3.5 Implementace optimalizačního modelu 56
4.3.6 Implementace databázové vrstvy 60
4.4 Použité nástroje při vývoji a testování 61
4.4.1 Github 61
4.4.2 Postman 61
4.4.3 Node.js a framework Express 61
4.4.4 Neo4j AuraDB 62
4.4.5 Jest a Supertest 62

5 Výsledky a diskuse 63
5.1 Vyhodnocení navrženého řešení 63
5.1.1 Vyhodnocení optimalizačního modelu 63
5.1.2 Kombinace automatického a manuálního přístupu 64
5.1.3 Přínosy využití grafové databáze 64
5.1.4 Celkové zhodnocení 64
5.2 Ověření řešení pomocí testovacího scénáře 65
5.2.1 Popis testovacího scénáře 65
5.2.2 Průběh výpočtu rozmístění 66
5.2.3 Vyhodnocení výsledného rozmístění 67
5.3 Možnosti dalšího rozvoje systému 70

6 Závěr 72

7 Seznam použitých zdrojů 73

8 Seznam obrázků, tabulek, grafů a zkratek 75
8.1 Seznam obrázků 75
8.2 Seznam tabulek 75
8.3 Seznam použitých zkratek 75

Přílohy..... 76

1 Úvod

Tato diplomová práce se zaměřuje na návrh a implementaci backendového řešení, jehož cílem je optimalizovat rozmístění jednotlivých oddělení v organizační struktuře firmy s využitím grafové databáze. V současné době se organizace stále častěji potýkají s potřebou efektivního řízení interních zdrojů a optimalizace pracovního prostředí. Jedním z praktických problémů, se kterým se mohou větší společnosti setkat, je rozmístění jednotlivých oddělení do dostupných budov a pater. Toto rozhodování musí zohledňovat řadu faktorů, mezi které patří například kapacitní omezení jednotlivých prostor, velikost jednotlivých oddělení nebo požadavky na vzájemnou blízkost oddělení, která spolu úzce spolupracují. V praxi je tento proces často řešen manuálně, což může být časově náročné a zároveň nemusí vždy vést k optimálnímu výsledku.

S rozvojem informačních technologií se stále více prosazují systémy, které dokážou podobné problémy řešit automatizovaně a s využitím pokročilých metod práce s daty. V tomto kontextu mohou být zajímavým řešením grafové databáze, které umožňují přirozeně modelovat vztahy mezi jednotlivými entitami systému. Na rozdíl od tradičních relačních databází jsou grafové databáze navrženy tak, aby efektivně pracovaly s propojenými daty a umožňovaly analyzovat složité vztahové struktury.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této diplomové práce je navrhnout a implementovat backendové řešení pro aplikaci, která usnadní rozmístění jednotlivých oddělení firmy do konkrétních pater a budov na základě definovaných kapacitních a organizačních požadavků. Systém umožní nastavit kapacity budov a jejich jednotlivých pater, velikosti oddělení, a zároveň definovat vzájemné vazby mezi odděleními na základě úrovně spolupráce. Součástí řešení bude možnost pevně přiřadit vybraná oddělení k určitým lokacím. Klíčovým cílem je využít grafovou databázi Neo4j pro efektivní modelování a vyhodnocování vztahů mezi jednotlivými entitami a implementovat algoritmus pro optimalizované rozmístění oddělení dle zadaných parametrů. Backendová aplikace bude realizována v prostředí `Node.js`

2.2 Metodika

První část práce bude popisovat přehled nástrojů a technik použitých v druhé části práce. Druhá část práce bude softwarová dokumentace aplikace popsané v cíli práce od fáze popisu požadavků až do pilotního provozu. Součástí práce bude také stručný uživatelský návod a popis použitých algoritmů.

3 Teoretická část práce

3.1 Informační systémy pro podporu rozhodování

Informační systémy představují v moderních organizacích klíčový nástroj pro zpracování, ukládání a analýzu dat, které podporují každodenní provoz i strategické rozhodování managementu. Jejich hlavním cílem je poskytovat uživatelům relevantní informace ve správný čas a ve vhodné podobě tak, aby mohli efektivně řídit procesy v organizaci a reagovat na změny v jejím okolí (Laudon, Laudon, 2020).

S rostoucí digitalizací podnikových procesů se zvyšuje množství dostupných dat, která mohou být využita při rozhodování. Samotná dostupnost dat však není dostatečná. Organizace potřebují nástroje, které umožňují tato data analyzovat, interpretovat a využít při řešení konkrétních problémů. Právě z tohoto důvodu vznikla kategorie informačních systémů označovaných jako systémy pro podporu rozhodování, známé jako *Decision Support Systems* (Laudon, Laudon, 2020).

Systémy pro podporu rozhodování jsou navrženy tak, aby pomáhaly uživatelům řešit složitější rozhodovací úlohy, které nelze snadno řešit pomocí jednoduchých pravidel nebo rutinních postupů. Tyto systémy kombinují práci s daty, analytické nástroje a modely, které umožňují analyzovat různé scénáře řešení daného problému DSS systémy často umožňují simulovat různé varianty řešení a porovnávat jejich dopady, což uživatelům usnadňuje výběr nejvhodnější alternativy (Laudon, Laudon, 2020).

Typickým příkladem využití systémů pro podporu rozhodování jsou úlohy spojené s plánováním zdrojů, optimalizací procesů nebo alokací kapacit. V těchto situacích je nutné zohlednit více různých faktorů současně, například dostupné zdroje, omezení systému a vztahy mezi jednotlivými prvky řešeného problému. Informační systém může v takovém případě sloužit jako nástroj, který tyto faktory systematicky zpracuje a umožní uživateli pracovat s různými variantami řešení (Techtarget, 2025).

Problematika plánování rozmístění organizačních jednotek v rámci budov společnosti představuje typickou rozhodovací úlohu tohoto typu. Při návrhu rozmístění oddělení je nutné současně zohlednit kapacitní omezení jednotlivých prostor, velikost jednotlivých oddělení a také jejich vzájemné vztahy a potřebu spolupráce. Při větším počtu oddělení se jedná o problém s rostoucí komplexitou, jehož řešení pouze pomocí manuálních postupů může být časově náročné a náchylné k chybám. V takové situaci může informační systém sloužit jako nástroj, který umožní systematicky evidovat relevantní data, modelovat

vztahy mezi jednotlivými entitami a podporovat proces hledání vhodného řešení. Díky využití vhodného datového modelu a algoritmického zpracování je možné automatizovat část rozhodovacího procesu a současně umožnit uživateli pracovat s různými variantami rozmístění (Techtarget, 2025).

3.2 Modelování dat v informačních systémech

Datové modelování představuje jeden ze základních kroků při návrhu informačních systémů. Jeho cílem je vytvořit strukturovanou reprezentaci dat, která budou v systému ukládána, zpracovávána a využívána při jednotlivých operacích aplikace. Správně navržený datový model umožňuje efektivní ukládání dat, jejich konzistentní správu a také snadnou interpretaci vztahů mezi jednotlivými prvky systému (IBM, 2022a).

Datový model lze obecně definovat jako abstraktní popis struktury dat, jejich vlastností a vzájemných vztahů. V rámci datového modelování jsou identifikovány jednotlivé entity systému, jejich atributy a vztahy mezi nimi. Tento proces umožňuje převést reálné objekty a procesy z řešené domény do formální struktury, která může být následně implementována v databázovém systému (IBM, 2022a).

Proces modelování dat obvykle začíná konceptuální úrovní, která popisuje základní objekty systému bez ohledu na konkrétní technologii ukládání dat. Na této úrovni jsou definovány například entity reprezentující objekty reálného světa, jejich vlastnosti a vztahy mezi nimi. Konceptuální model slouží jako základ pro další fáze návrhu databáze a umožňuje lépe porozumět struktuře řešeného problému (IBM, 2022a).

Na konceptuální model navazuje logický datový model, který již více reflektuje konkrétní způsob organizace dat v databázovém systému. V této fázi se definují například primární klíče, vazby mezi entitami nebo struktura jednotlivých atributů. Logický model tak představuje přechod mezi abstraktním popisem dat a jejich konkrétní implementací v databázi (IBM, 2022a).

Poslední úrovní je fyzický datový model, který popisuje konkrétní implementaci dat v databázovém systému. V této fázi se řeší například způsob ukládání dat, indexace, optimalizace dotazů nebo struktura databázových tabulek či grafových struktur. Fyzický model je tedy již přímo svázán s konkrétní databázovou technologií.

V rámci návrhu informačních systémů se používá několik různých přístupů k modelování dat. Historicky nejrozšířenějším přístupem je relační datový model, který reprezentuje data ve formě tabulek propojených prostřednictvím vztahů mezi primárními

a cizími klíči. Tento model se stal standardem v oblasti databázových systémů a je využíván ve většině tradičních podnikových aplikací.

S rozvojem nových typů aplikací, které pracují s velkým množstvím propojených dat, se však začaly prosazovat také alternativní přístupy k ukládání dat. Jedním z nich je grafový datový model, který reprezentuje data pomocí uzlů a vztahů mezi nimi. Tento přístup umožňuje přirozeně reprezentovat komplexní vztahy mezi jednotlivými objekty a je proto vhodný zejména pro aplikace, kde hraje významnou roli struktura vztahů mezi daty (IBM, 2022a).

Volba vhodného datového modelu má zásadní vliv na způsob implementace aplikace, výkon databázových operací i možnosti dalšího rozšiřování systému. V případě aplikací, které pracují s komplexními vztahy mezi entitami, může grafový přístup poskytovat výrazné výhody oproti tradičnímu relačnímu modelu. Z tohoto důvodu je v následujících kapitolách věnována pozornost jak relačnímu datovému modelu, tak zejména grafovým databázím, které představují základní technologii využitou při implementaci navrhovaného systému (IBM, 2022a).

3.2.1 Relační datový model

Relační datový model představuje jeden z nejrozšířenějších způsobů organizace dat v databázových systémech. Tento model byl poprvé formálně popsán v roce 1970 matematikem Edgarem F. Coddem a od té doby se stal základem většiny tradičních databázových systémů využívaných v informačních systémech.

Základním principem relačního datového modelu je reprezentace dat ve formě tabulek, které jsou tvořeny řádky a sloupci. Každá tabulka reprezentuje určitou entitu reálného světa, například zaměstnance, oddělení nebo budovy. Jednotlivé řádky tabulky představují konkrétní záznamy a sloupce definují atributy dané entity (IBM, 2022b).

Každá tabulka obsahuje primární klíč, který jednoznačně identifikuje jednotlivé záznamy. Primární klíč je obvykle tvořen jedním nebo více atributy, jejichž hodnota je pro každý záznam unikátní. Kromě primárního klíče mohou tabulky obsahovat také cizí klíče, které slouží k vytváření vztahů mezi jednotlivými tabulkami. Pomocí těchto klíčů je možné propojit data uložená v různých tabulkách a vytvářet tak komplexnější struktury dat.

Relační databázové systémy využívají pro práci s daty dotazovací jazyk SQL, který umožňuje data ukládat, upravovat, vyhledávat a mazat. SQL poskytuje

standardizovaný způsob práce s databázemi a je podporován většinou moderních databázových systémů, jako jsou například MySQL, PostgreSQL nebo Oracle Database.

Důležitým principem návrhu relačních databází je také normalizace dat. Normalizace představuje proces organizace databázových tabulek tak, aby byla minimalizována redundance dat a zároveň byla zachována jejich konzistence. Tento proces zahrnuje rozdělování dat do více tabulek a definování vztahů mezi nimi pomocí primárních a cizích klíčů (IBM, 2022b).

Relační datový model je vhodný zejména pro aplikace, které pracují s dobře strukturovanými daty a kde jsou vztahy mezi entitami relativně stabilní. Tento model umožňuje efektivní správu dat a poskytuje silné mechanismy pro zajištění integrity databáze. Na druhou stranu může být práce s velmi komplexními vztahy mezi velkým množstvím entit v relačních databázích složitější, protože vyžaduje využití více spojení tabulek prostřednictvím operací typu JOIN (IBM, 2022b).

Právě omezení spojená s reprezentací složitých vztahových struktur vedla v posledních letech k rozvoji alternativních databázových modelů, mezi které patří například grafové databáze. Ty umožňují reprezentovat vztahy mezi objekty přirozeněji a často efektivněji pracovat s daty, která mají charakter rozsáhlých propojených sítí (IBM, 2022b).

3.2.2 Grafový datový model

Grafový datový model představuje způsob strukturování dat založený na teorii grafů. Základními prvky tohoto modelu jsou uzly a vztahy, které mezi sebou vytvářejí síť propojených entit. Tento přístup umožňuje přirozeně reprezentovat data, u nichž hrají klíčovou roli vazby mezi jednotlivými objekty (Robinson, Webber, Eifrem, 2015).

Uzly reprezentují jednotlivé entity v systému, například osoby, oddělení nebo budovy. Každý uzel může obsahovat sadu vlastností, které blíže popisují jeho charakteristiky. Vztahy pak vyjadřují spojení mezi uzly a mají vždy definovaný směr. Stejně jako uzly mohou i vztahy obsahovat vlastnosti, které specifikují povahu daného spojení.

Důležitou vlastností grafového datového modelu je jeho flexibilita. Na rozdíl od tradičních relačních modelů není nutné předem definovat pevné schéma databáze. Struktura dat se může přirozeně vyvíjet podle aktuálních potřeb aplikace, což umožňuje snadnější rozšiřování modelu o nové typy uzlů nebo vztahů (Robinson, Webber, Eifrem, 2015).

Grafový model se zaměřuje především na efektivní reprezentaci a procházení vztahů mezi entitami. Díky tomu je možné rychle analyzovat komplexní propojení, aniž by bylo nutné provádět složité operace nad více tabulkami. Tento přístup je vhodný zejména pro domény, kde jsou vztahy mezi daty stejně důležité nebo důležitější než samotná data.

Formálně lze grafový datový model popsat jako orientovaný graf, kde množina uzlů a množina hran vytváří strukturu umožňující efektivní navigaci mezi jednotlivými prvky. Tento princip tvoří základ pro fungování grafových databází, které jsou schopny takto strukturovaná data ukládat a efektivně zpracovávat (Robinson, Webber, Eifrem, 2015; Bruggen, 2025).

3.3 Grafové databáze

Grafové databáze představují typ databázových systémů, které jsou navrženy pro ukládání a zpracování dat ve formě grafové struktury. Na rozdíl od tradičních relačních databází, které ukládají data do tabulek, grafové databáze pracují s uzly reprezentujícími jednotlivé objekty a vztahy mezi nimi. Tento přístup umožňuje přirozeně modelovat komplexní propojení mezi daty a efektivně analyzovat jejich vzájemné vazby (Robinson, Webber, Eifrem, 2015).

3.3.1 Oblasti využití

Grafové databáze nacházejí uplatnění především v oblastech, kde je klíčové efektivně pracovat se vztahy mezi entitami a analyzovat jejich vzájemné propojení. Díky schopnosti rychlého procházení vazeb jsou vhodné pro řešení problémů, které jsou v relačních databázích obtížněji realizovatelné.

Jednou z nejčastějších oblastí využití jsou sociální sítě, kde je potřeba modelovat vztahy mezi uživateli, jako jsou přátelství, sledování nebo interakce. Grafové databáze umožňují efektivně vyhledávat například společné kontakty, doporučovat nové spojení nebo analyzovat strukturu celé sítě (Lyon, 2022).

Další významnou oblastí jsou doporučovací systémy. Ty využívají informace o chování uživatelů a jejich vazbách na produkty či služby k navrhování relevantního obsahu. Grafová reprezentace umožňuje snadno identifikovat podobnosti mezi uživateli nebo položkami a generovat doporučení na základě těchto vztahů.

Grafové databáze se uplatňují také v oblasti detekce podvodů, například ve finančním sektoru. Pomocí analýzy transakčních vazeb mezi účty lze odhalovat podezřelé vzory chování, které by v tradičních databázích nebyly snadno identifikovatelné.

Další oblastí je správa a analýza sítí, například v telekomunikacích nebo dopravních systémech. Grafový přístup umožňuje modelovat infrastrukturu jako síť propojených uzlů a optimalizovat například tok dat nebo dopravy (Lyon, 2022).

V neposlední řadě lze grafové databáze využít v oblasti plánování a optimalizace, kde je nutné zohlednit vztahy mezi jednotlivými prvky systému. Typickým příkladem je optimalizace rozmístění zdrojů nebo organizačních jednotek s ohledem na jejich vzájemnou blízkost a vazby (Lyon, 2022).

3.3.2 Výhody oproti relačním databázím

Grafové databáze přinášejí oproti relačním databázím odlišný přístup k ukládání a zpracování dat, který je výhodný zejména v situacích, kde hrají klíčovou roli vztahy mezi entitami. Zatímco relační databáze ukládají data do tabulek a vazby mezi nimi realizují pomocí cizích klíčů a operací JOIN, grafové databáze ukládají vztahy přímo jako součást datové struktury (IBM 2022b).

Jednou z hlavních výhod grafových databází je efektivní práce s propojenými daty. Při dotazování není nutné provádět složité spojování tabulek, což může být u relačních databází výpočetně náročné, zejména při větším počtu vazeb. Grafové databáze umožňují přímé procházení vztahů mezi uzly, což vede k rychlejšímu získání výsledků při analýze komplexních struktur (IBM 2022b).

Další výhodou je flexibilita datového modelu. Relační databáze vyžadují předem definované schéma, jehož změna může být náročná a časově nákladná. Naproti tomu grafové databáze umožňují snadnější rozšiřování struktury dat bez nutnosti zásadních úprav existujícího modelu (Robinson, Webber, Eifrem, 2015).

Významným přínosem je také přehlednost a přirozenost reprezentace dat. Vztahy mezi entitami jsou v grafové databázi explicitně vyjádřeny, což usnadňuje pochopení struktury dat i návrh dotazů. Tento přístup je vhodný zejména pro modelování reálných scénářů, kde jsou vazby mezi objekty zásadní.

Rozdíly mezi oběma přístupy se projevují i při práci s neúplnými nebo volitelnými vztahy. V relačních databázích je často nutné použít operace typu LEFT JOIN, aby byly zahrnuty i záznamy bez odpovídajících vazeb. Grafové databáze umožňují obdobnou

funkcionalitu přirozeně pomocí konstrukcí jako OPTIONAL MATCH, díky čemuž jsou i nepropojené uzly zahrnuty do výsledků bez potřeby složitějších dotazů (IBM 2022b; Robinson, Webber, Eifrem, 2015).

3.3.3 Neo4j

Neo4j je jednou z nejrozšířenějších grafových databází, která implementuje grafový datový model a je navržena pro efektivní práci s propojenými daty. Jedná se o nativní grafovou databázi, což znamená, že ukládá data přímo ve formě uzlů a vztahů, nikoli jako jejich abstrakci nad jinou datovou strukturou (Robinson, Webber, Eifrem, 2015).

Pro práci s databází Neo4j se používá dotazovací jazyk Cypher, který je navržen tak, aby umožňoval intuitivní práci s grafovou strukturou. Dotazy v jazyce Cypher využívají zápis založený na vzorech, který umožňuje jednoduše specifikovat požadované uzly a vztahy mezi nimi. Tento přístup usnadňuje formulaci dotazů zaměřených na procházení a analýzu vztahů.

Neo4j je dostupná jak ve formě lokální instalace, tak jako cloudová služba Neo4j AuraDB. Cloudové řešení eliminuje potřebu správy infrastruktury a umožňuje soustředit se především na návrh datového modelu a implementaci aplikační logiky.

Výhodou Neo4j je také široká podpora nástrojů a integrací, například s backendovými technologiemi jako `Node.js`, což umožňuje snadné napojení databáze na aplikační vrstvu. Díky tomu je možné efektivně implementovat systémy, které využívají grafový přístup pro řešení komplexních problémů (Bruggen, 2025; Lyon, 2022).

3.4 Architektura webových aplikací

Moderní informační systémy jsou často realizovány ve formě webových aplikací, které umožňují přístup k datům a funkcím systému prostřednictvím webového rozhraní. Tyto aplikace jsou zpravidla navrženy na základě architektonických principů, které určují způsob komunikace mezi jednotlivými částmi systému. Jedním z nejrozšířenějších přístupů je klient–server architektura.

3.4.1 Klient – server architektura

Webové aplikace jsou obvykle navrženy podle principu klient–server architektury, ve které jsou jednotlivé části systému rozděleny mezi klientskou a serverovou část.

V rámci klient–server architektury vystupuje klient jako aplikace, která komunikuje se serverem a zobrazuje uživatelské rozhraní. Serverová část aplikace zajišťuje zpracování požadavků klienta, práci s databází a implementaci aplikační logiky systému. Tento přístup umožňuje oddělit prezentační vrstvu aplikace od logiky zpracování dat, což usnadňuje vývoj, údržbu i další rozšiřování systému (Laudon, Laudon, 2020; GeeksforGeeks, 2026a).

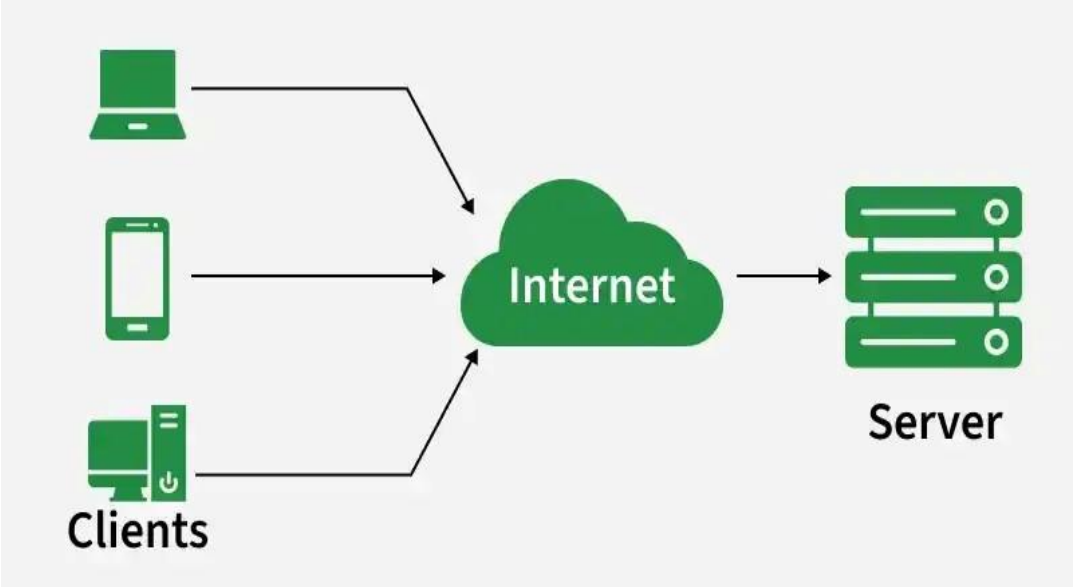
V současných webových aplikacích je běžné rozdělení systému na frontend a backend. Frontend představuje klientskou část aplikace, která je zodpovědná za prezentaci dat uživateli a interakci s uživatelským rozhraním. Backend naopak obsahuje aplikační logiku systému a zajišťuje komunikaci s databází, zpracování požadavků a poskytování dat klientské části aplikace.

Komunikace mezi klientem a serverem probíhá prostřednictvím síťových protokolů, nejčastěji pomocí protokolu HTTP. Klient zasílá serveru požadavky obsahující informace o požadované operaci a server na tyto požadavky odpovídá vrácením požadovaných dat nebo výsledku zpracování. Tento mechanismus umožňuje oddělit jednotlivé části aplikace a vytvářet modulární systémy, které lze snadno rozšiřovat a integrovat s dalšími službami (GeeksforGeeks, 2026a).

Důležitou součástí moderních webových aplikací je také využití aplikačních programových rozhraní neboli Application Programming Interface. API představuje rozhraní, které umožňuje komunikaci mezi jednotlivými částmi systému nebo mezi různými aplikacemi. V případě webových aplikací se často používají tzv. webová API, která umožňují klientským aplikacím přistupovat k datům a funkcím backendového systému.

Architektura založená na oddělení klientské a serverové části přináší několik významných výhod. Jednou z nich je možnost nezávislého vývoje jednotlivých částí aplikace. Frontend i backend mohou být vyvíjeny samostatně, případně i různými vývojovými týmy. Další výhodou je možnost využití různých technologií pro jednotlivé části systému podle jejich konkrétních požadavků.

Architektura webových aplikací založená na oddělení klientské a serverové části tak umožňuje vytvořit flexibilní a rozšiřitelný systém, který lze v budoucnu dále rozvíjet nebo integrovat s dalšími informačními systémy (Laudon, Laudon, 2020; GeeksforGeeks, 2026a).



Obrázek 1: Client – server model (GeeksforGeeks, 2026a)

3.4.2 REST architektura

Representational State Transfer představuje architektonický styl určený pro návrh distribuovaných systémů a webových služeb. Tento koncept byl poprvé popsán Royem Fieldingem ve své disertační práci a stal se jedním z nejpoužívanějších přístupů při návrhu webových API. REST architektura je založena na využití standardních webových technologií a protokolu HTTP pro komunikaci mezi klientem a serverem.

Základním principem REST architektury je práce s prostředky, které jsou identifikovány pomocí jednoznačných adres ve formě URI. Každý prostředek reprezentuje určitou entitu systému, například uživatele, produkt nebo jiný objekt. Klientská aplikace může s těmito prostředky pracovat prostřednictvím standardních HTTP metod (Fielding 2000; MDN Web Docs, 2025a).

Mezi nejčastěji používané HTTP metody patří metoda GET, která slouží k získání dat ze serveru, metoda POST určená pro vytvoření nového záznamu, metoda PUT využívaná pro aktualizaci existujících dat a metoda DELETE, která slouží k odstranění záznamu. Tyto metody umožňují implementovat základní operace nad daty známé jako CRUD operace, tedy Create, Read, Update a Delete (MDN Web Docs, 2025a).

Dalším důležitým principem REST architektury je bezstavovost komunikace. Tento princip znamená, že každý požadavek odeslaný klientem obsahuje všechny informace

potřebné k jeho zpracování a server si neuchovává informace o předchozích požadavcích. Díky tomu je možné serverovou aplikaci snadněji škálovat a rozšiřovat.

REST architektura také klade důraz na jednotné rozhraní, které definuje standardizovaný způsob komunikace mezi klientem a serverem. Použití jednotného rozhraní zjednodušuje integraci různých systémů a umožňuje snadnější vývoj klientských aplikací (Fielding, 2000).

Webové služby založené na REST architektuře jsou často implementovány jako tzv. REST API. Takové API poskytuje sadu endpointů, prostřednictvím kterých mohou klientské aplikace přistupovat k datům a funkcím serverové aplikace. Endpoint představuje konkrétní adresu, na kterou klient zasílá požadavky pro získání nebo úpravu dat.

REST API se stalo standardním přístupem při návrhu moderních webových aplikací, protože umožňuje jednoduchou a přehlednou komunikaci mezi jednotlivými částmi systému. Díky využití standardních webových technologií je možné REST služby snadno integrovat s různými typy klientských aplikací, jako jsou webové aplikace, mobilní aplikace nebo jiné informační systémy (MDN Web Docs, 2025a).

3.5 Backendové technologie

Backendová část webových aplikací představuje serverovou vrstvu systému, která zajišťuje zpracování požadavků klientských aplikací, práci s databází a implementaci aplikační logiky systému. Backend přijímá požadavky od klientské části aplikace, zpracovává je a následně vrací odpověď obsahující požadovaná data nebo výsledek provedené operace (MDN Web Docs, 2025b).

Serverová část aplikace obvykle obsahuje několik základních komponent. Mezi ně patří například aplikační logika systému, která zpracovává požadavky klienta, dále vrstva komunikující s databází a také rozhraní ve formě API, které umožňuje přístup k jednotlivým funkcím systému. Tyto komponenty společně tvoří backendovou architekturu aplikace (MDN Web Docs, 2025b).

Při vývoji backendových aplikací lze využít celou řadu různých programovacích jazyků a technologických platforem. Mezi často používané technologie patří například Java, Python, PHP nebo JavaScript. Výběr konkrétní technologie závisí na požadavcích projektu, dostupných nástrojích a zkušenostech vývojového týmu.

V posledních letech se stále více prosazuje využití prostředí `Node.js`, které umožňuje vývoj serverových aplikací v programovacím jazyce JavaScript. Tento přístup umožňuje využívat stejný programovací jazyk jak na straně klienta, tak na straně serveru, což může zjednodušit vývoj a údržbu aplikace.

Backendové aplikace jsou často realizovány pomocí různých aplikačních frameworků, které poskytují základní strukturu aplikace a usnadňují implementaci běžných funkcí, jako je práce s HTTP požadavky, směrování endpointů nebo zpracování dat. Tyto frameworky umožňují vývojářům soustředit se na implementaci aplikační logiky a zjednodušují vývoj komplexnějších systémů (MDN Web Docs, 2025b).

3.5.1 JavaScript

JavaScript je skriptovací programovací jazyk, který byl původně navržen pro vývoj webových aplikací na straně klienta. Jazyk byl vytvořen v roce 1995 Brendanem Eichem ve společnosti Netscape a postupně se stal jedním z nejrozšířenějších programovacích jazyků používaných při vývoji webových aplikací. JavaScript je dnes standardizován organizací ECMA International a jeho specifikace je známá pod názvem ECMAScript.

Původně byl JavaScript využíván především pro implementaci interaktivních prvků v prostředí webového prohlížeče. Postupem času se však jeho využití výrazně rozšířilo a dnes se používá nejen na straně klienta, ale také na straně serveru. Díky tomu je možné využívat jeden programovací jazyk pro vývoj celé webové aplikace, což může zjednodušit vývoj i údržbu systému (MDN Web Docs, 2025c).

JavaScript je dynamicky typovaný programovací jazyk, který podporuje různá programovací paradigmata, zejména objektově orientované a funkcionální programování. Jazyk umožňuje pracovat s funkcemi jako s plnohodnotnými objekty a podporuje práci s událostmi a asynchronní zpracování operací. Tyto vlastnosti jsou důležité zejména při vývoji webových aplikací, kde je často potřeba zpracovávat více operací současně. Významnou vlastností jazyka JavaScript je také podpora asynchronního programování. Asynchronní zpracování umožňuje provádět operace, například komunikaci se serverem nebo práci s databází, bez blokování běhu aplikace. V moderních aplikacích se k tomu využívají například konstrukce jako Promise nebo klíčová slova async a await (MDN Web Docs, 2025c).

Rozvoj prostředí Node.js umožnil využití jazyka JavaScript také pro vývoj serverových aplikací. Díky tomu se JavaScript stal univerzálním nástrojem pro vývoj webových aplikací, který lze použít jak pro implementaci uživatelského rozhraní, tak pro implementaci backendové logiky systému (MDN Web Docs, 2025c).

3.5.2 Node.js

Node.js představuje runtime prostředí určené pro spouštění aplikací napsaných v programovacím jazyce JavaScript na straně serveru. Díky prostředí Node.js je možné využívat jazyk JavaScript nejen v prostředí webového prohlížeče, ale také při implementaci serverových aplikací, které zpracovávají požadavky klientských aplikací a komunikují s databázemi nebo dalšími službami (Node.js, 2023).

Node.js je postaven na JavaScriptovém enginu V8, který byl původně vyvinut společností Google pro webový prohlížeč Chrome. Tento engine umožňuje efektivní spouštění JavaScriptového kódu a poskytuje vysoký výkon při zpracování serverových operací. Díky tomu je Node.js vhodný zejména pro aplikace, které vyžadují rychlé zpracování velkého množství požadavků (Node.js, 2023).

Jednou z klíčových vlastností Node.js je jeho asynchronní a událostmi řízený model zpracování požadavků. Tento model umožňuje efektivně pracovat s operacemi vstupu a výstupu, například při komunikaci s databází nebo při práci se soubory. Namísto blokujícího zpracování jednotlivých operací Node.js využívá neblokující I/O operace, které umožňují serveru zpracovávat více požadavků současně (Node.js, 2023).

Díky tomuto přístupu je Node.js schopen efektivně obsluhovat velké množství klientských požadavků bez nutnosti vytváření velkého počtu paralelních vláken. Tento model je vhodný zejména pro aplikace poskytující webové služby nebo API rozhraní, kde je nutné rychle zpracovávat velké množství požadavků od různých klientských aplikací.

Node.js poskytuje také rozsáhlý ekosystém knihoven a nástrojů dostupných prostřednictvím správce balíčků npm. Tento ekosystém umožňuje vývojářům využívat širokou škálu existujících modulů, které usnadňují implementaci různých funkcí serverové aplikace, například práci s HTTP požadavky, autentizaci uživatelů nebo komunikaci s databázemi (Node.js, 2023).

Díky své flexibilitě a výkonu se Node.js stal jednou z nejpoužívanějších technologií pro vývoj backendových aplikací a webových API. Je využíván v celé řadě moderních

webových služeb a aplikací, které vyžadují vysoký výkon a schopnost zpracovávat velké množství současných požadavků (Node.js, 2023).

3.5.3 Express

`Express` je open-source webový framework pro prostředí `Node.js`, který slouží k vývoji webových aplikací a serverových API. Tento framework poskytuje jednoduchý a flexibilní způsob vytváření serverových aplikací, které zpracovávají HTTP požadavky a komunikují s klientskými aplikacemi. `Express` patří mezi nejpoužívanější frameworky pro `Node.js` a je často využíván při implementaci REST API.

Jednou z hlavních funkcí frameworku `Express` je zpracování HTTP požadavků a směrování jednotlivých požadavků na odpovídající části aplikace. Tento mechanismus je označován jako routing a umožňuje definovat jednotlivé endpointy aplikace. Každý endpoint je spojen s konkrétní funkcí, která zpracovává požadavek a vrací odpověď klientovi (MDN Web Docs, 2025d).

`Express` podporuje práci s různými HTTP metodami, například GET, POST, PUT nebo DELETE, které odpovídají základním operacím nad daty v systému. Díky tomu je možné pomocí `Expressu` jednoduše implementovat REST API, které poskytuje přístup k datům uloženým v databázi (Taha, 2017)..

Framework `Express` také umožňuje využívat tzv. middleware funkce. Middleware představuje funkce, které se vykonávají během zpracování HTTP požadavku a mohou provádět různé operace, například kontrolu autentizace uživatele, zpracování vstupních dat nebo logování požadavků. Tento mechanismus umožňuje modulární strukturování aplikace a usnadňuje implementaci různých funkcí serverové logiky (Node.js, 2023).

Další výhodou frameworku `Express` je jeho jednoduchost a flexibilita. Framework poskytuje pouze základní nástroje pro práci s HTTP komunikací a nechává vývojářům velkou volnost při návrhu struktury aplikace. Díky tomu je možné `Express` využít jak pro jednoduché webové aplikace, tak pro komplexnější serverové systémy (Node.js, 2023).

`Express` je často využíván ve spojení s dalšími knihovnami a nástroji, které rozšiřují jeho funkcionalitu. Patří mezi ně například knihovny pro práci s databázemi, validaci vstupních dat nebo správu autentizace uživatelů. Díky rozsáhlému ekosystému nástrojů je možné pomocí `Expressu` vytvářet plnohodnotné backendové aplikace (Taha, 2017).

3.6 Optimalizační úlohy a lineární programování

3.6.1 Optimalizační úlohy s omezeními

Optimalizační úlohy představují důležitou oblast operačního výzkumu, matematiky a informatiky. Jejich cílem je nalezení takového řešení daného problému, které je z určitého hlediska nejlepší, přičemž je nutné respektovat předem definovaná omezení. Tato omezení vycházejí z reálných podmínek systému, ve kterém je problém řešen, například z kapacitních limitů, organizačních pravidel nebo technologických možností (Taha, 2017; Encyclopeadia Britanica, 2023).

V obecné podobě lze optimalizační úlohu formulovat jako problém hledání hodnot rozhodovacích proměnných, které maximalizují nebo minimalizují určitou účelovou funkci. Účelová funkce vyjadřuje kritérium, podle kterého se jednotlivá řešení porovnávají. Může například reprezentovat maximalizaci efektivity, minimalizaci nákladů nebo optimalizaci využití dostupných zdrojů. Vedle účelové funkce je model tvořen také soustavou omezení, která definují podmínky, jež musí každé přípustné řešení splňovat (Taha, 2017).

Optimalizační úlohy se v praxi objevují v celé řadě oblastí. Patří mezi ně například plánování výroby, řízení logistických procesů, plánování dopravy, alokace zdrojů nebo plánování rozvrhů. V informačních systémech se optimalizační přístupy často využívají při řešení problémů, kde je potřeba nalézt efektivní konfiguraci systému při respektování určitých omezení (Encyclopeadia Britanica, 2023).

Jedním z typických příkladů je problém rozmístění organizačních jednotek nebo týmů v rámci fyzických prostor organizace. Při takovém rozhodování je nutné zohlednit nejen kapacitu jednotlivých prostor, ale také organizační vztahy mezi jednotlivými odděleními. Vhodné rozmístění může výrazně ovlivnit efektivitu komunikace, spolupráce mezi týmy i celkovou produktivitu organizace (Encyclopeadia Britanica, 2023).

Cílem optimalizační úlohy je nalézt takové rozmístění oddělení, které respektuje kapacitní omezení jednotlivých pater a zároveň podporuje umístění spolupracujících oddělení co nejblíže k sobě. Tento typ problému je možné formálně popsat pomocí matematického modelu, který obsahuje rozhodovací proměnné reprezentující umístění jednotlivých oddělení, kapacitní omezení prostor a optimalizační kritérium vyjadřující vhodnost výsledného rozmístění. Výsledkem řešení optimalizační úlohy je taková konfigurace systému, která nejlépe splňuje definovaná pravidla a optimalizační kritéria (Taha, 2017).

3.6.2 Kombinatorická optimalizace

Problém rozmístění oddělení do dostupných prostor lze zařadit mezi úlohy kombinatorické optimalizace. Tento typ optimalizačních úloh se vyznačuje tím, že hledané řešení je tvořeno výběrem vhodné kombinace prvků z diskrétní množiny možností. Na rozdíl od úloh spojitých optimalizačních metod, kde mohou proměnné nabývat libovolných reálných hodnot, zde mají rozhodovací proměnné typicky diskrétní charakter, například reprezentují rozhodnutí typu „ano/ne“ nebo „přiřazeno / nepřiřazeno“ (Encyclopeadia Britanica, 2023).

Kombinatorické optimalizační problémy se často vyznačují velmi rychlým růstem počtu možných řešení. Tento jev je způsoben tím, že každé nové rozhodnutí přidává další možné kombinace řešení. Například při rozmísťování většího počtu oddělení do několika pater existuje velké množství variant jejich přiřazení. Pokud by například bylo potřeba rozmístit několik desítek oddělení do více pater různých budov, počet všech možných konfigurací by mohl být velmi vysoký. Prohledávání všech těchto možností by proto bylo výpočetně velmi náročné (Encyclopeadia Britanica, 2023).

Z tohoto důvodu se při řešení kombinatorických optimalizačních úloh využívají různé optimalizační přístupy, které umožňují efektivně prohledávat prostor možných řešení. Tyto metody využívají matematické modely nebo algoritmy, které systematicky vyhodnocují jednotlivé varianty řešení a postupně hledají takové řešení, které splňuje definovaná omezení a zároveň optimalizuje zvolenou účelovou funkci.

Kombinatorická optimalizace je využívána v celé řadě praktických aplikací. Typickými příklady jsou například problémy plánování tras v dopravě, rozvrhování výrobních procesů, plánování rozvrhů ve školství, alokace zdrojů nebo návrh logistických sítí. Společným znakem těchto problémů je nutnost nalézt vhodnou konfiguraci systému z velkého množství možných variant.

V kontextu této práce představuje kombinatorický problém úloha přiřazení oddělení k jednotlivým patrům budov. Každé oddělení může být umístěno pouze na jedno patro a každé patro má omezenou kapacitu. Současně je vhodné zohlednit vztahy mezi odděleními, zejména jejich spolupráci, která může ovlivňovat vhodnost výsledného rozmístění. Optimalizační model proto musí nalézt takovou kombinaci přiřazení oddělení k jednotlivým patrům, která splňuje kapacitní omezení a současně podporuje vhodné prostorové uspořádání organizace (Encyclopeadia Britanica, 2023).

Kombinatorická povaha problému znamená, že počet potenciálních řešení může být velmi vysoký. Z tohoto důvodu je vhodné využít formalizovaný optimalizační model, který umožňuje systematicky hledat řešení splňující všechna omezení a současně optimalizující definované kritérium kvality výsledného rozmístění (Encyclopeadia Britanica, 2023).

3.6.3 Lineární programování

Jedním z nejvýznamnějších nástrojů pro řešení optimalizačních úloh je lineární programování. Jedná se o matematickou metodu, která umožňuje nalézt optimální hodnoty rozhodovacích proměnných při maximalizaci nebo minimalizaci lineární účelové funkce za předpokladu lineárních omezení. Lineární programování patří mezi základní nástroje operačního výzkumu a je široce využíváno v mnoha praktických aplikacích, například při plánování výroby, řízení zásob, optimalizaci dopravy nebo při alokaci zdrojů.

Matematický model lineárního programování je obvykle tvořen třemi základními složkami. První z nich jsou rozhodovací proměnné, které reprezentují jednotlivá rozhodnutí v rámci řešeného problému. Tyto proměnné vyjadřují například množství vyráběných produktů, objem přepravovaného zboží nebo přiřazení určitého zdroje k určitému úkolu (Taha, 2017; Encyclopeadia Britanica, 2023).

Druhou složkou modelu je účelová funkce, která vyjadřuje optimalizační kritérium. Tato funkce může například reprezentovat maximalizaci zisku, minimalizaci nákladů nebo maximalizaci efektivity využití zdrojů. Účelová funkce je lineární kombinací rozhodovacích proměnných, přičemž jednotlivé koeficienty vyjadřují relativní význam jednotlivých proměnných.

Třetí složkou modelu jsou omezení, která definují podmínky, jež musí řešení splňovat. Omezení jsou rovněž vyjádřena lineárními rovnicemi nebo nerovnicemi. V praxi mohou představovat například kapacitní limity, technologická omezení nebo organizační pravidla (Encyclopeadia Britanica, 2023).

Výsledkem řešení modelu lineárního programování je taková kombinace hodnot rozhodovacích proměnných, která splňuje všechna omezení a současně optimalizuje hodnotu účelové funkce. Pro řešení těchto úloh existuje řada efektivních algoritmů, mezi nejznámější patří například simplexová metoda nebo metody založené na vnitřních bodech.

Lineární programování se v praxi často využívá při řešení problémů alokace zdrojů, kde je potřeba rozhodnout, jakým způsobem rozdělit omezené zdroje mezi jednotlivé

aktivity. V případě problémů, kde rozhodovací proměnné reprezentují spojité hodnoty, poskytuje lineární programování velmi efektivní nástroj pro nalezení optimálního řešení.

V některých praktických situacích však rozhodovací proměnné nemohou nabývat libovolných reálných hodnot. Často je nutné, aby reprezentovaly diskrétní rozhodnutí, například zda je určitý objekt přiřazen k určitému zdroji. V takových případech je vhodné využít rozšíření lineárního programování známé jako celočíselné lineární programování, které umožňuje pracovat s celočíselnými nebo binárními proměnnými (Taha, 2017).

3.6.4 Celočíselné lineární programování

V mnoha praktických optimalizačních úlohách není možné, aby rozhodovací proměnné nabývaly libovolných reálných hodnot. Často je nutné, aby reprezentovaly diskrétní rozhodnutí, například zda je určitý objekt přiřazen k určitému zdroji nebo zda je určitá varianta řešení zvolena či nikoliv. V takových případech se využívá celočíselné lineární programování, anglicky *Integer Linear Programming*.

Celočíselné lineární programování představuje rozšíření klasického lineárního programování, ve kterém musí některé nebo všechny rozhodovací proměnné nabývat pouze celočíselných hodnot. Speciálním případem tohoto přístupu je binární lineární programování, kde proměnné mohou nabývat pouze hodnot 0 nebo 1. Tento typ proměnných je vhodný zejména pro modelování rozhodnutí typu „ano“ nebo „ne“ (Taha, 2017; MIT, 2010).

Binární proměnné jsou velmi často využívány v úlohách přiřazení, kde reprezentují informaci, zda je určitý prvek přiřazen k určitému zdroji. Například při řešení problému rozmístění oddělení může binární proměnná vyjadřovat, zda je konkrétní oddělení umístěno na určité patro budovy. Hodnota proměnné je rovna 1 v případě, že je oddělení na daném patře umístěno, a 0 v případě, že na daném patře umístěno není (Taha, 2017).

Použití celočíselného lineárního programování umožňuje vytvořit formální matematický model optimalizační úlohy, který zahrnuje rozhodovací proměnné, účelovou funkci a soustavu omezení. Omezení mohou například zajišťovat, že každé oddělení je přiřazeno maximálně na jedno patro nebo že součet velikostí oddělení umístěných na daném patře nepřekročí jeho kapacitu.

Řešení celočíselných optimalizačních úloh je obecně výpočetně náročnější než řešení klasických úloh lineárního programování, protože prostor možných řešení je diskrétní. Pro

jejich řešení se proto využívají specializované algoritmy, mezi které patří například metoda větví a mezí nebo její různá rozšíření. Tyto algoritmy umožňují systematicky prohledávat prostor možných řešení a nalézt optimální řešení, které splňuje všechna definovaná omezení.

Celočíselné lineární programování je v praxi využíváno například při plánování výroby, optimalizaci logistických sítí, návrhu dopravních systémů nebo při alokaci zdrojů v organizačních strukturách. Díky možnosti modelovat diskrétní rozhodnutí je tento přístup vhodný také pro problémy přiřazení objektů k omezeným kapacitám (Taha, 2017; MIT, 2010).

3.7 Modelování softwarových systémů

Při návrhu a vývoji softwarových systémů je důležité využívat vhodné metody modelování, které umožňují přehledně popsat strukturu systému, jeho jednotlivé komponenty a vzájemné vztahy mezi nimi. Modelování softwarových systémů slouží k lepšímu pochopení navrhovaného řešení, k usnadnění komunikace mezi vývojáři a dalšími účastníky projektu a zároveň také k dokumentaci návrhu systému (IBM, 2023c).

Modelování umožňuje zachytit jak statickou strukturu systému, tak i jeho dynamické chování. Pomocí různých typů modelů lze popsat například jednotlivé entity systému, jejich vlastnosti, vztahy mezi nimi nebo jednotlivé procesy a interakce uživatelů se systémem. Modely tak představují abstraktní reprezentaci reálného systému, která pomáhá snížit jeho komplexitu a umožňuje soustředit se na klíčové aspekty návrhu.

V oblasti softwarového inženýrství existuje řada metod a nástrojů určených pro modelování systémů. Mezi nejrozšířenější patří modelovací jazyk Unified Modeling Language, který poskytuje standardizovaný způsob grafického znázornění různých aspektů softwarového systému (IBM, 2023c).

Použití modelovacích technik je běžnou součástí procesu návrhu softwarových systémů. Pomáhá nejen při samotném vývoji aplikace, ale také při její dokumentaci a budoucím rozšiřování (IBM, 2023c).

3.7.1 UML

Unified Modeling Language je standardizovaný modelovací jazyk určený pro vizuální reprezentaci a dokumentaci softwarových systémů. Jazyk UML byl vyvinut v 90. letech minulého století a stal se jedním z nejpoužívanějších nástrojů pro modelování

v oblasti softwarového inženýrství. Jeho cílem je poskytnout jednotný způsob, jak graficky znázornit strukturu systému, jeho chování a interakce mezi jednotlivými komponentami.

UML umožňuje vytvářet různé typy diagramů, které popisují různé aspekty systému. Tyto diagramy lze obecně rozdělit do dvou hlavních skupin – strukturální diagramy a behaviorální diagramy. Strukturální diagramy slouží k popisu statické struktury systému, například tříd, objektů nebo vztahů mezi jednotlivými entitami. Mezi nejčastěji používané strukturální diagramy patří například diagram tříd nebo diagram komponent (IBM, 2023c).

Behaviorální diagramy naopak popisují dynamické chování systému a interakce mezi jeho jednotlivými částmi. Tyto diagramy umožňují znázornit například procesy probíhající v systému, komunikaci mezi objekty nebo interakci uživatele se systémem. Mezi běžně používané behaviorální diagramy patří například diagram případů užití, sekvenční diagram nebo diagram aktivit (IBM, 2023c).

Výhodou UML je jeho standardizace a široká podpora v nástrojích pro modelování softwaru. Díky tomu je možné UML diagramy snadno sdílet mezi členy vývojového týmu a využívat je jako součást dokumentace systému. UML diagramy rovněž usnadňují komunikaci mezi vývojáři, analytiky a dalšími účastníky projektu, protože poskytují přehledný a srozumitelný způsob znázornění návrhu systému (IBM, 2023c).

3.7.2 Diagram případů užití

Diagram případů užití, v anglické terminologii označovaný jako use case diagram, patří mezi behaviorální diagramy modelovacího jazyka UML a slouží k popisu funkcionalit systému z pohledu jeho uživatelů. Hlavním účelem tohoto diagramu je znázornit, jaké služby nebo funkce systém poskytuje jednotlivým aktérům a jakým způsobem s nimi mohou uživatelé interagovat (IBM, 2023c; Lucid Software, 2026).

Základními prvky diagramu případů užití jsou aktéři, případy užití a vztahy mezi nimi. Aktér představuje entitu, která se systémem komunikuje. Může se jednat například o uživatele systému, administrátora nebo jiný externí systém. Aktéři nejsou součástí samotného systému, ale představují jeho okolí, které s ním vstupuje do interakce.

Případ užití představuje konkrétní funkci nebo službu, kterou systém poskytuje aktérovi. Každý případ užití popisuje určitý scénář interakce mezi uživatelem a systémem. Může se jednat například o vytvoření nového záznamu, zobrazení dat, úpravu existujících

informací nebo spuštění určitého procesu v systému. Případy užití jsou v diagramu obvykle znázorňovány pomocí elips, zatímco aktéři jsou reprezentováni stylizovanou postavou.

Mezi jednotlivými prvky diagramu mohou existovat různé typy vztahů. Nejčastěji se jedná o vztah asociace mezi aktérem a případem užití, který vyjadřuje, že aktér může danou funkcionalitu systému využívat. Dalšími vztahy mohou být například vztahy *include* nebo *extend*, které umožňují modelovat závislosti mezi jednotlivými případy užití (IBM, 2023c).

Diagram případů užití slouží především k zachycení funkčních požadavků systému na vysoké úrovni. Pomáhá identifikovat hlavní funkcionality systému a zároveň poskytuje přehled o tom, jakým způsobem budou uživatelé se systémem pracovat. Díky své přehlednosti je tento diagram často využíván v počátečních fázích návrhu systému, kdy je potřeba definovat základní rozsah funkcionality aplikace (Lucid Software, 2026).

3.7.3 Diagram tříd

Diagram tříd patří mezi strukturální diagramy modelovacího jazyka UML a slouží k popisu statické struktury softwarového systému. Pomocí diagramu tříd je možné znázornit jednotlivé třídy systému, jejich vlastnosti, metody a také vztahy mezi jednotlivými třídami. Tento diagram představuje jeden z nejdůležitějších nástrojů při návrhu objektově orientovaných systémů.

Každá třída v diagramu představuje určitou entitu systému a obsahuje informace o jejích vlastnostech a operacích. V diagramu je třída obvykle znázorněna jako obdélník rozdělený do několika částí. V horní části je uveden název třídy, prostřední část obsahuje seznam atributů a spodní část může obsahovat seznam metod, které třída poskytuje. Diagram tříd tak umožňuje přehledně zobrazit strukturu dat a funkcionalitu jednotlivých částí systému (IBM, 2023c; IBM 2023d).

Mezi třídami mohou existovat různé typy vztahů, které vyjadřují jejich vzájemné vazby. Nejčastěji se jedná o asociaci, která reprezentuje základní vztah mezi dvěma třídami. Dalším typem vztahu je například dědičnost, která umožňuje vytvářet hierarchii tříd a znovupoužívat společné vlastnosti. V UML se také využívají vztahy agregace a kompozice, které vyjadřují vztah celek–část mezi jednotlivými třídami.

Diagram tříd je často využíván při návrhu datového modelu aplikace a při návrhu struktury zdrojového kódu. Umožňuje vývojářům lépe porozumět organizaci systému a usnadňuje komunikaci mezi členy vývojového týmu. Zároveň slouží jako důležitá součást technické dokumentace softwarového projektu (IBM, 2023c).

3.8 Testování webových API

Testování softwaru představuje nedílnou součást procesu vývoje informačních systémů. Jeho hlavním cílem je ověřit, zda navržený a implementovaný systém odpovídá stanoveným požadavkům, vykazuje očekávané chování a je dostatečně spolehlivý pro reálné použití (Myers, Sandler, Badgett, 2021). V případě backendových aplikací, které poskytují funkcionalitu prostřednictvím aplikačního rozhraní, nabývá testování ještě většího významu, protože právě API představuje hlavní komunikační vrstvu mezi serverovou částí systému a klientskými aplikacemi.

Webové API umožňuje klientské části aplikace přistupovat k datům a funkcím backendu prostřednictvím standardizovaných HTTP požadavků. Každý endpoint systému tak představuje konkrétní vstupní bod, přes který lze vyvolat určitou operaci, například načtení dat, vytvoření nového záznamu, úpravu existující entity nebo odstranění dat ze systému (Myers, Sandler, Badgett, 2021). Správná funkčnost těchto endpointů je z pohledu backendové aplikace zásadní, protože chybná implementace API může způsobit nefunkčnost celé aplikace i v případě, že ostatní části systému byly navrženy správně (Myers, Sandler, Badgett, 2021).

Testování webových API se zaměřuje na ověření toho, zda server správně přijímá požadavky, správně zpracovává vstupní data a vrací odpovídající výstupy. Při testování se obvykle sleduje několik základních aspektů. Prvním z nich je samotná dostupnost a funkčnost endpointů, tedy zda systém reaguje na požadavky očekávaným způsobem. Dalším důležitým hlediskem je správnost vrácených dat, která bývají v moderních webových aplikacích nejčastěji přenášena ve formátu JSON. Testování dále zahrnuje i kontrolu stavových kódů HTTP, které informují o výsledku zpracování požadavku a umožňují klientské části aplikace správně interpretovat odpověď serveru.

Významnou součástí testování API je rovněž ověřování chování systému v nestandardních nebo chybových situacích. Nestačí totiž pouze potvrdit, že aplikace funguje správně při korektních vstupech, ale je také nutné ověřit, jak systém reaguje při neúplných, neplatných nebo neočekávaných datech. Právě tyto situace často odhalují nedostatky v implementaci validační logiky, ošetření chyb nebo celkové robustnosti backendové aplikace (Myers, Sandler, Badgett, 2021).

Z metodického hlediska lze testování webových API rozdělit na manuální a automatizované. Manuální testování je vhodné zejména v průběhu vývoje a ladění systému, kdy je potřeba rychle ověřovat chování konkrétních endpointů a analyzovat odpovědi serveru. Automatizované testování naopak umožňuje opakovaně ověřovat správnou funkčnost backendu při každé změně zdrojového kódu a přispívá tak k dlouhodobé stabilitě systému. Oba přístupy mají v procesu vývoje své opodstatnění a v praxi se často vzájemně doplňují (GeeksforGeeks, 2026b).

3.8.1 Manuální testování API

Manuální testování aplikačního rozhraní představuje základní přístup k ověřování funkčnosti webových API během vývoje softwaru. Tento typ testování spočívá v ručním odesílání HTTP požadavků na serverovou aplikaci a následné analýze odpovědí, které server vrací. Vývojář nebo tester tak může postupně ověřovat jednotlivé endpointy aplikace a kontrolovat, zda backend reaguje očekávaným způsobem (GeeksforGeeks, 2026b).

Při manuálním testování API se obvykle testují jednotlivé operace dostupné prostřednictvím HTTP metod. Nejčastěji se jedná o metody GET, POST, PUT nebo DELETE, které odpovídají základním operacím práce s daty. Metoda GET slouží k načítání dat ze systému, metoda POST k vytváření nových záznamů, metoda PUT k aktualizaci existujících dat a metoda DELETE k jejich odstranění. Při testování je důležité ověřit nejen správnou funkčnost těchto operací, ale také správnost vrácených dat a odpovídající stavové kódy HTTP (GeeksforGeeks, 2026b).

Kromě samotného obsahu odpovědi serveru je při testování API důležité sledovat také strukturu vrácených dat. Moderní webové aplikace obvykle využívají formát JSON, který umožňuje přehledný a standardizovaný přenos dat mezi klientskou a serverovou částí aplikace. Tester tak může kontrolovat, zda vrácená data obsahují očekávané atributy a zda mají správnou datovou strukturu.

Pro manuální testování API existuje řada specializovaných nástrojů, které umožňují vytvářet a odesílat HTTP požadavky bez nutnosti implementace klientské aplikace. Jedním z nejpoužívanějších nástrojů je Postman. Tento nástroj poskytuje uživatelské rozhraní, ve kterém je možné definovat jednotlivé požadavky, nastavovat parametry, hlavičky nebo tělo požadavku a následně analyzovat odpověď serveru (GeeksforGeeks, 2026b).

Významnou funkcí nástroje Postman je možnost organizovat jednotlivé požadavky do tzv. kolekcí. Kolekce představují logické skupiny API požadavků, které mohou odpovídat

jednotlivým částem aplikace nebo konkrétním funkčním celkům systému. Díky tomu je možné testovací scénáře přehledně organizovat a opakovaně využívat při dalším vývoji aplikace. Kolekce mohou být také sdíleny mezi členy vývojového týmu, což usnadňuje spolupráci při testování backendových služeb.

Manuální testování API je zvláště užitečné v počátečních fázích vývoje aplikace, kdy je potřeba rychle ověřovat správnou implementaci jednotlivých endpointů a ladit chování systému. Vývojář může postupně testovat jednotlivé scénáře komunikace mezi klientem a serverem a kontrolovat, zda aplikace reaguje správně na různé typy vstupních dat (GeeksforGeeks, 2026b).

3.8.2 Automatizované testování API

Automatizované testování představuje přístup k testování softwaru, při kterém jsou testovací scénáře spouštěny automaticky pomocí specializovaných nástrojů a testovacích frameworků. Na rozdíl od manuálního testování, které vyžaduje přímou interakci vývojáře nebo testera, umožňuje automatizované testování opakované a systematické ověřování správné funkčnosti aplikace bez nutnosti ručního provádění jednotlivých testovacích kroků.

V moderním vývoji softwaru hraje automatizované testování významnou roli, protože umožňuje rychle identifikovat chyby vzniklé při změnách zdrojového kódu. Testy mohou být spouštěny například při každé úpravě aplikace nebo při nasazení nové verze systému. Díky tomu je možné zajistit, že nové změny neovlivní již existující funkcionalitu aplikace. Tento přístup bývá často označován jako regresní testování (GeeksforGeeks, 2026b).

Automatizované testy backendových aplikací se často zaměřují na ověřování správné funkčnosti jednotlivých API endpointů. Testovací scénáře mohou simulovat HTTP požadavky, které běžně odesílá klientská část aplikace, a následně kontrolovat odpovědi serveru. Při testování se ověřují například stavové kódy HTTP, struktura vrácených dat nebo správnost zpracování vstupních parametrů.

Významnou výhodou automatizovaného testování je jeho opakovatelnost a rychlost. Testy lze spouštět opakovaně bez dodatečných nákladů na lidskou práci a mohou být integrovány do vývojového procesu aplikace. Díky tomu mohou vývojáři průběžně ověřovat stabilitu systému a minimalizovat riziko zavedení nových chyb do již funkční části aplikace.

Automatizované testování také přispívá ke zlepšení kvality zdrojového kódu. Přítomnost testů často vede vývojáře k vytváření přehlednějších a modulárnějších částí aplikace, které je možné snadno testovat. Testy tak mohou sloužit nejen jako nástroj pro kontrolu funkčnosti systému, ale také jako forma dokumentace chování jednotlivých částí aplikace (GeeksforGeeks, 2025c).

V případě backendových aplikací implementovaných v prostředí `Node.js` se pro automatizované testování často využívají specializované testovací frameworky a knihovny, které umožňují simulovat HTTP komunikaci a ověřovat chování serverové aplikace. Mezi často používané nástroje patří například testovací framework `Jest` a knihovna Supertest, které umožňují vytvářet automatizované testy pro REST API aplikací (GeeksforGeeks, 2026b).

3.8.3 Testovací framework Jest

`Jest` je moderní testovací framework určený pro aplikace napsané v programovacím jazyce JavaScript. Framework byl vyvinut společností Meta a v současnosti patří mezi nejpoužívanější nástroje pro testování aplikací v prostředí `Node.js`. `Jest` poskytuje komplexní prostředí pro vytváření, organizaci a spouštění automatizovaných testů a je navržen tak, aby umožňoval jednoduché a přehledné definování testovacích scénářů.

Jednou z hlavních výhod frameworku `Jest` je jeho jednoduchá konfigurace a snadná integrace do projektů využívajících JavaScript. `Jest` je často označován jako tzv. „*zero-configuration*“ nástroj, protože v základní podobě nevyžaduje složité nastavení a je možné jej rychle začlenit do existujícího projektu. Díky tomu mohou vývojáři začít vytvářet testy bez nutnosti rozsáhlé konfigurace testovacího prostředí (Meta Platforms, Inc., 2025).

Testy ve frameworku `Jest` jsou obvykle organizovány do tzv. testovacích sad, které obsahují jednotlivé testovací případy. Testovací případ představuje konkrétní scénář, který ověřuje určitou funkčnost aplikace. Každý test obsahuje definici očekávaného výsledku a následné porovnání tohoto očekávání se skutečným chováním systému. Tento mechanismus je realizován pomocí tzv. *asercí*, které umožňují kontrolovat různé vlastnosti výsledků testů (Meta Platforms, Inc., 2025).

`Jest` poskytuje také přehledné výstupy testování, které informují o průběhu a výsledcích jednotlivých testů. Vývojáři tak mohou snadno identifikovat části aplikace,

ve kterých se vyskytují chyby. Framework rovněž umožňuje spouštět testy paralelně, což zvyšuje rychlost testování i u větších projektů.

Další výhodou frameworku `Jest` je jeho schopnost pracovat s asynchronním kódem, který je v prostředí `Node.js` velmi běžný. `Jest` umožňuje testovat asynchronní operace například při komunikaci s databází nebo při zpracování HTTP požadavků. Díky tomu je možné spolehlivě ověřovat funkčnost backendových aplikací využívajících asynchronní zpracování dat (Meta Platforms, Inc., 2025).

3.8.4 Knihovna Supertest

Supertest je knihovna určená pro testování HTTP serverů v prostředí `Node.js`. Nejčastěji se používá ve spojení s testovacími frameworky, jako je například `Jest`, a umožňuje jednoduše vytvářet automatizované testy pro webové aplikace poskytující REST API. Hlavním účelem této knihovny je simulovat HTTP požadavky na serverovou aplikaci a následně analyzovat odpovědi, které server vrací (Ladjs, 2024).

Pomocí knihovny Supertest je možné testovat jednotlivé endpointy aplikace bez nutnosti vytváření samostatné klientské aplikace. Testovací skript může například odeslat HTTP požadavek na konkrétní endpoint serveru a následně ověřit, zda server vrátil očekávaný stavový kód HTTP a odpovídající data. Tento přístup umožňuje efektivně testovat chování backendové aplikace a ověřovat správnost implementace jednotlivých funkcí systému (GeeksforGeeks, 2025; Ladjs, 2024).

Supertest umožňuje pracovat s různými HTTP metodami, které jsou běžně využívány v REST API, například GET, POST, PUT nebo DELETE. Testy mohou ověřovat nejen samotný stavový kód odpovědi serveru, ale také strukturu vrácených dat nebo hodnoty jednotlivých atributů v odpovědi. Díky tomu je možné přesně definovat očekávané chování jednotlivých endpointů aplikace.

Velkou výhodou knihovny Supertest je její jednoduchá integrace s testovacími frameworky. Ve spojení s frameworkem `Jest` lze vytvářet testovací scénáře, které simulují komunikaci mezi klientem a serverem a zároveň automaticky vyhodnocují výsledky jednotlivých testů. Tento přístup umožňuje ověřovat správnou funkčnost backendové aplikace při každé změně zdrojového kódu (Ladjs, 2024).

Automatizované testy využívající knihovnu Supertest mohou například kontrolovat správné zpracování požadavků na vytvoření nebo úpravu dat, validaci vstupních parametrů

nebo chování aplikace při chybových stavech. Testování tak pomáhá odhalit chyby v implementaci a zvyšuje celkovou spolehlivost systému (GeeksforGeeks, 2025c).

3.9 Verzovací systémy

Při vývoji softwaru je běžné, že na projektu pracuje více vývojářů a že se zdrojový kód postupně vyvíjí a mění. Pro správu těchto změn se využívají verzovací systémy, které umožňují sledovat historii úprav zdrojového kódu, ukládat jednotlivé verze projektu a usnadňovat spolupráci mezi členy vývojového týmu.

Verzovací systém zaznamenává jednotlivé změny provedené v souborech projektu a umožňuje se kdykoliv vrátit k předchozí verzi kódu. Díky tomu je možné snadno identifikovat, kdy byla určitá změna provedena, kdo ji provedl a jaký měla obsah (Chacon, Straub, 2014). Verzovací systémy také umožňují pracovat s více verzemi projektu současně, což je důležité například při vývoji nových funkcí nebo při opravách chyb.

Moderní verzovací systémy často podporují distribuovaný způsob práce, kdy každý vývojář pracuje s vlastní kopií repozitáře a změny jsou následně synchronizovány s centrálním úložištěm. Tento přístup umožňuje efektivnější spolupráci mezi vývojáři a zároveň zvyšuje bezpečnost uložených dat (Chacon, Straub, 2014).

Použití verzovacích systémů je dnes standardní součástí vývoje softwaru a je běžné jak v profesionálním vývoji, tak i v akademických projektech. Verzovací systémy přispívají k lepší organizaci projektu, umožňují sledovat vývoj aplikace v čase a usnadňují správu zdrojového kódu (Chacon, Straub, 2014).

3.9.1 Git

Git je distribuovaný verzovací systém určený pro správu zdrojového kódu softwarových projektů. Byl vytvořen v roce 2005 Linusem Torvaldsem při vývoji operačního systému Linux a postupně se stal jedním z nejpoužívanějších verzovacích systémů v oblasti softwarového vývoje (Chacon, Straub, 2014).

Základním principem systému Git je ukládání historie změn v repozitáři, který obsahuje kompletní historii vývoje projektu. Každá změna v projektu je zaznamenána ve formě tzv. commitu, který představuje konkrétní stav projektu v určitém okamžiku.

Commit obsahuje informace o provedených změnách, autorovi změny a čase jejího vytvoření.

Git umožňuje vytvářet větve projektu, které slouží k paralelnímu vývoji různých částí aplikace. Vývojáři mohou například vytvářet samostatné větve pro implementaci nových funkcí nebo pro testování změn, aniž by ovlivnili stabilní verzi projektu. Po dokončení práce je možné tyto změny sloučit zpět do hlavní větve projektu.

Díky distribuované povaze systému Git má každý vývojář k dispozici kompletní kopii repozitáře včetně celé historie projektu. To umožňuje pracovat s verzemi projektu i bez připojení k internetu a zároveň poskytuje vyšší bezpečnost dat, protože historie projektu není uložena pouze na jednom centrálním serveru (Chacon, Straub, 2014).

3.9.2 GitHub

GitHub je webová platforma určená pro hostování repozitářů systému Git a pro spolupráci při vývoji softwaru. Poskytuje prostředí, ve kterém mohou vývojáři ukládat své projekty, sdílet zdrojový kód a spolupracovat na jeho dalším rozvoji.

Platforma GitHub umožňuje ukládání repozitářů v cloudu a poskytuje řadu nástrojů pro správu projektů. Uživatelé mohou sledovat historii změn v projektu, spravovat jednotlivé větve, vytvářet požadavky na sloučení změn nebo diskutovat o navrhovaných úpravách kódu. Díky těmto funkcím je GitHub často využíván při týmovém vývoji softwaru (Coursera, 2025).

Další výhodou platformy GitHub je možnost veřejného nebo soukromého sdílení repozitářů. Veřejné repozitáře jsou přístupné široké komunitě vývojářů a umožňují sdílení open-source projektů. Soukromé repozitáře jsou naopak přístupné pouze vybraným uživatelům a využívají se například při vývoji interních nebo akademických projektů (Coursera, 2025).

4 Vlastní práce

4.1 Analýza zadání

Tato kapitola se zabývá analýzou požadavků na navrhovaný systém. Definuje vstupy, omezení a základní pravidla, která budou následně zohledněna při návrhu architektury a implementaci backendového řešení.

4.1.1 Vznik požadavku a návrh řešení

Impulsem pro vznik navrhované aplikace je plánované stěhování společnosti do nových administrativních prostor. Společnost bude nově disponovat několika budovami s rozdílným počtem pater a omezenou kapacitou jednotlivých podlaží. Součástí procesu stěhování je rozhodnutí o tom, jak budou jednotlivá oddělení rozmístěna tak, aby bylo zajištěno efektivní využití dostupného prostoru a současně byla podpořena jejich vzájemná spolupráce.

Zadavatelem požadavku je zaměstnanec společnosti, který je pověřen koordinací procesu stěhování a návrhem nového prostorového uspořádání organizačních jednotek. V rámci přípravné fáze bylo zjištěno, že při absenci specializovaného nástroje by bylo nutné celý proces řešit manuálně, zejména prostřednictvím tabulkových procesorů a ručního porovnávání kapacitních omezení jednotlivých pater a organizačních vazeb mezi odděleními. Tento přístup je sice technicky realizovatelný, avšak při větším počtu oddělení se stává nepřehledným, časově náročným a náchylným k chybám.

Manuální práce s tabulkami navíc neumožňuje systematicky vyhodnocovat kvalitu jednotlivých variant rozmístění ani jednoduše porovnávat více scénářů. Rozhodování je v takovém případě založeno převážně na subjektivním odhadu a opakovaném přepočítávání kapacit, což může vést k neoptimálním výsledkům. V situaci, kdy je nutné současně zohlednit kapacitní limity jednotlivých pater, velikost oddělení vycházející z počtu zaměstnanců, pevně definovaná umístění vybraných týmů a vazby spolupráce mezi odděleními, se problém stává kombinatoricky složitým.

Na základě této potřeby vznikl návrh na vytvoření specializované aplikace, která umožní formalizovat vstupní parametry, modelovat vztahy mezi jednotlivými entitami a automatizovaně generovat návrh rozmístění oddělení respektující definovaná omezení. Součástí návrhu bylo také využití grafové databáze pro efektivní reprezentaci vztahů mezi budovami, patry, odděleními a zaměstnanci.

Dále bude implementován mechanismus pro automatický výpočet návrhu rozmístění oddělení. Tento výpočet je založen na optimalizačním modelu, který zohledňuje kapacitní omezení jednotlivých pater, velikost oddělení i definované vztahy spolupráce mezi odděleními. Výsledkem výpočtu bude návrh rozmístění, který splňuje stanovená omezení a současně bude usilovat o co nejvhodnější prostorové uspořádání organizačních jednotek.

Navržená aplikace tak nahradí manuální tabulkový přístup systematickým, opakovatelným a transparentním procesem, který umožní generovat a porovnávat různé varianty rozmístění oddělení v rámci dostupných kancelářských prostor.

4.1.2 Funkční požadavky systému

Funkční požadavky definují konkrétní funkcionalitu, kterou musí navrhovaný systém poskytovat. V případě navrhované aplikace se jedná především o evidenci organizačních jednotek, správu prostorové struktury budov a podporu automatizovaného návrhu rozmístění oddělení v rámci dostupných kancelářských prostor.

Na základě analýzy problému byly identifikovány následující funkční požadavky systému.

FP1 – Správa budov
Systém musí umožňovat evidenci jednotlivých budov, které jsou k dispozici pro umístění oddělení. U každé budovy musí být možné uložit její identifikaci a počet pater.

FP2 – Správa pater
Systém musí umožňovat evidovat jednotlivá patra v rámci budov. Každé patro musí být jednoznačně přiřazeno ke konkrétní budově a musí být možné definovat jeho kapacitu, která určuje maximální počet zaměstnanců nebo oddělení, jež mohou být na daném patře umístěni.

FP3 – Správa oddělení
Systém musí umožňovat evidenci jednotlivých oddělení organizace. Každé oddělení představuje organizační jednotku, která má být v rámci procesu stěhování umístěna do konkrétní budovy a patra.

FP4 – Správa zaměstnanců

Systém musí umožňovat evidenci zaměstnanců a jejich přiřazení k jednotlivým oddělením. Počet zaměstnanců přiřazených k oddělení slouží jako základ pro určení velikosti oddělení, která je následně zohledněna při návrhu rozmístění.

FP5 – Evidence spolupráce mezi odděleními

Systém musí umožňovat definovat vazby spolupráce mezi jednotlivými odděleními. Tyto vazby vyjadřují potřebu prostorové blízkosti oddělení a jsou zohledňovány při výpočtu výsledného rozmístění.

FP6 – Automatický výpočet rozmístění oddělení

Systém musí umožňovat spustit automatický výpočet návrhu rozmístění oddělení do jednotlivých budov a pater. Výpočet musí zohledňovat kapacitní omezení pater, velikost oddělení a definované vazby spolupráce mezi odděleními.

FP7 – Zobrazení aktuálního rozmístění

Systém musí umožňovat zobrazit aktuální rozmístění oddělení v rámci jednotlivých budov a pater. Uživatel musí mít přehled o tom, která oddělení jsou umístěna na konkrétním patře a v jaké budově.

FP8 – Manuální úprava rozmístění

Systém musí umožňovat provádět manuální změny rozmístění oddělení, například přesunutí oddělení na jiné patro nebo do jiné budovy. Tyto změny mohou být prováděny i po automatickém výpočtu rozmístění.

4.1.3 Nefunkční požadavky

Nefunkční požadavky definují vlastnosti systému, které se netýkají přímo konkrétní funkcionality, ale určují kvalitu, provozní vlastnosti a technické omezení navrhovaného řešení. Tyto požadavky ovlivňují zejména výkon systému, jeho architekturu, rozšiřitelnost a způsob provozu.

Na základě analýzy zadání byly identifikovány následující nefunkční požadavky.

NP1 – Výkonnost systému
Systém musí být schopen generovat návrh rozmístění oddělení v přijatelném čase i při větším množství vstupních dat. Výpočet rozmístění musí být proveden tak, aby uživatel nemusel čekat nepřiměřeně dlouhou dobu na získání výsledného návrhu.

NP2 – Škálovatelnost řešení
Systém musí umožňovat práci s větším počtem budov, pater, oddělení a zaměstnanců. Navržené řešení musí být schopno zpracovávat rostoucí objem dat bez zásadních změn architektury systému.

NP3 – Modularita backendu
Backend aplikace musí být navržen modulárně tak, aby jednotlivé části systému byly odděleny podle své odpovědnosti. Díky tomu je možné jednotlivé moduly systému snadno upravovat, rozšiřovat nebo nahrazovat bez zásadního zásahu do ostatních částí aplikace.

NP4 – Testovatelnost systému
Architektura backendu musí umožňovat automatizované testování jednotlivých částí aplikace. API rozhraní systému musí být navrženo tak, aby bylo možné jeho funkčnost ověřovat pomocí automatizovaných testů.

NP5 – Přenositelnost řešení
Systém musí být navržen tak, aby mohl být provozován na různých platformách bez nutnosti zásadních úprav zdrojového kódu. Použití prostředí `Node.js` a webového rozhraní REST API umožňuje provoz systému na různých operačních systémech.

NP6 – Bezpečnost přístupu k databázi

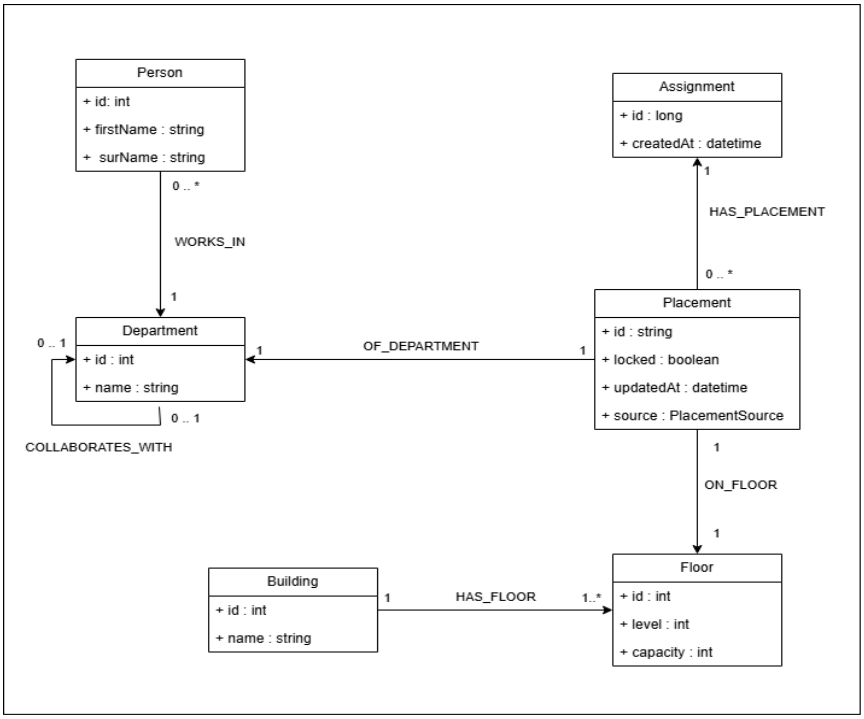
Přístup k databázi musí být realizován prostřednictvím konfiguračních proměnných a přihlašovacích údajů, které nejsou přímo součástí zdrojového kódu aplikace. Tím je zajištěno bezpečné oddělení citlivých údajů od aplikační logiky.

4.2 Návrh systému

4.2.1 Datový model systému

Na základě provedené analýzy požadavků byl navržen datový model systému, který popisuje základní entity aplikace a vztahy mezi nimi. Cílem návrhu bylo vytvořit takovou strukturu dat, která by umožňovala reprezentovat organizační strukturu společnosti, prostorové uspořádání budov a současně podporovala proces rozmísťování jednotlivých oddělení do dostupných prostor. Navržený model musel zároveň umožnit pracovat s více variantami rozmístění oddělení a podporovat kombinaci automatického výpočtu rozmístění s manuálními zásahy uživatele.

Při návrhu datového modelu byly identifikovány entity reprezentující zaměstnance, oddělení, budovy a jednotlivá patra. Tyto entity byly doplněny o prvky umožňující reprezentovat konkrétní umístění oddělení v prostoru a evidovat jednotlivé verze rozmístění v čase. Díky tomu bylo možné oddělit popis organizační struktury společnosti od konkrétních variant rozmístění oddělení, které mohou vznikat při plánování stěhování společnosti.



Obrázek 2: UML diagram tříd datového modelu systému (vlastní zpracování)

Person

Uzel typu Person reprezentuje jednotlivého zaměstnance společnosti. Každý zaměstnanec je přiřazen ke konkrétnímu oddělení. Počet zaměstnanců přiřazených k oddělení slouží jako základ pro určení velikosti oddělení, která je následně zohledněna při návrhu rozmístění.

Department

Uzel typu *Department* reprezentuje organizační jednotku společnosti. Oddělení představuje základní jednotku, která je v rámci procesu stěhování umisťována do konkrétní budovy a patra. Oddělení může mít definovanou vazbu spolupráce s jiným oddělením, což vyjadřuje požadavek na jejich prostorovou blízkost.

Building

Uzel typu *Building* reprezentuje jednotlivé budovy, které jsou k dispozici pro umístění oddělení. Každá budova může obsahovat více pater, která definují konkrétní prostorové jednotky využitelné pro rozmístění oddělení.

Floor

Uzel typu *Floor* reprezentuje jednotlivá patra budov. Každé patro má definovanou kapacitu, která určuje maximální počet zaměstnanců, jež mohou být na daném patře umístěni.

Assignment

Uzel typu *Assignment* představuje konkrétní variantu rozmístění oddělení. Každý výpočet rozmístění vytváří novou instanci tohoto uzlu, která slouží jako samostatná verze návrhu rozmístění.

Placement

Uzel typu *Placement* reprezentuje konkrétní umístění oddělení na určité patro v rámci dané varianty rozmístění. Tento uzel propojuje oddělení, patro a konkrétní variantu rozmístění.

Kromě samotných uzlů je datový model tvořen také vztahy, které definují propojení mezi jednotlivými entitami systému.

WORKS_IN

Vztah WORKS_IN propojuje uzel typu Person s uzlem typu *Department* a vyjadřuje příslušnost zaměstnance k danému oddělení.

LOCATED_IN

Vztah LOCATED_IN propojuje patro s budovou, ve které se dané patro nachází.

COOPERATES_WITH

Vztah COOPERATES_WITH vyjadřuje spolupráci mezi dvěma odděleními. Tento vztah je zohledňován při návrhu rozmístění a může ovlivnit výsledné umístění oddělení tak, aby spolupracující oddělení byla umístěna v prostorové blízkosti.

HAS_PLACEMENT

Vztah HAS_PLACEMENT propojuje variantu rozmístění s jednotlivými umístěními oddělení.

PLACED_ON

Vztah PLACED_ON propojuje oddělení s konkrétním patrem v rámci určité varianty rozmístění.

4.2.2 Diagram případů užití

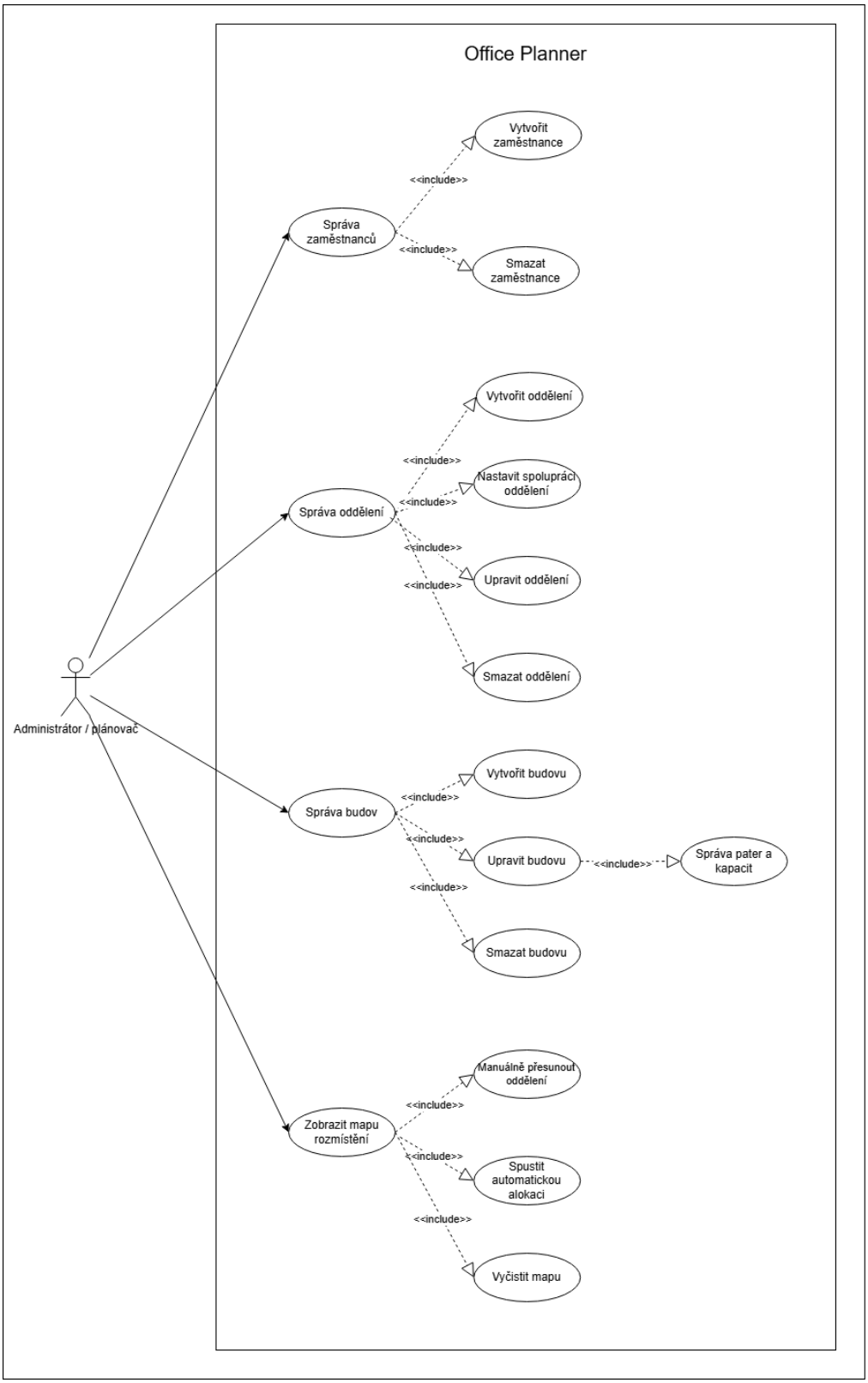
Na základě specifikace systému a identifikovaných funkčních požadavků byly definovány hlavní interakce uživatele se systémem. Tyto interakce byly formalizovány pomocí diagramu případů užití neboli use case diagramu, jehož cílem bylo zachytit základní funkce aplikace z pohledu uživatele a vymezit hranice systému. Diagram případů užití umožňuje přehledně znázornit, jaké operace systém poskytuje a jakým způsobem s ním uživatel pracuje při plánování rozmístění oddělení v rámci budov společnosti. Hlavním aktérem systému je uživatel vystupující v roli administrátora nebo plánovače, který je odpovědný za správu vstupních dat systému a za samotný proces návrhu rozmístění oddělení.

Jednu skupinu případů užití tvoří operace spojené se správou zaměstnanců. Uživatel může vytvářet nové zaměstnance, odstraňovat existující záznamy a přiřazovat zaměstnance k jednotlivým odděleními. Tyto operace slouží k udržování aktuálního seznamu zaměstnanců a současně definují organizační strukturu společnosti na úrovni jednotlivých týmů. Další skupinu případů užití představuje správa oddělení, v jejímž rámci může uživatel vytvářet nová oddělení, upravovat jejich vlastnosti nebo je ze systému odstraňovat. Součástí správy oddělení je také možnost definovat vazbu spolupráce mezi dvěma odděleními. Tato vazba vyjadřuje zvýšenou potřebu komunikace mezi odděleními a je následně zohledňována při procesu automatické alokace oddělení.

Další část funkcionality systému je zaměřena na správu budov a jejich prostorové struktury. Uživatel může vytvářet nové budovy, upravovat jejich názvy nebo je ze systému odstraňovat. Součástí této části systému je také správa jednotlivých pater a jejich kapacitních omezení, která určují maximální počet pracovních míst dostupných na daném patře. Tyto informace představují důležitý vstup pro algoritmus rozmísťování oddělení, protože definují prostorová omezení, se kterými musí systém při výpočtu pracovat.

Samostatnou oblast případů užití tvoří práce s mapou rozmístění oddělení. Uživatel může zobrazit aktuální mapu rozmístění, která znázorňuje umístění jednotlivých oddělení v rámci budov a jejich pater. V rámci této části systému je možné provádět manuální přesuny oddělení na konkrétní patra, čímž lze určit pevné umístění vybraných oddělení. Systém současně umožňuje spustit automatickou alokaci oddělení, která na základě aktuálních vstupních dat vypočítá nové rozmístění oddělení s ohledem na kapacitní omezení a vztahy mezi jednotlivými odděleními. Součástí této funkcionality je také možnost vyčištění mapy rozmístění, která vytvoří nový prázdný stav a umožní zahájit návrh rozmístění znovu.

Diagram případů užití znázorňující jednotlivé funkce systému je uveden na následujícím obrázku.



Obrázek 3: Diagram případů užití (vlastní zpracování)

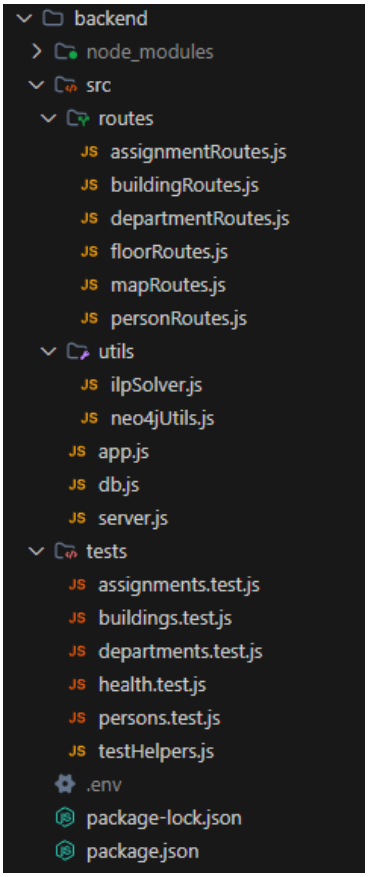
.

4.3 Implementace systému

4.3.1 Architektura backendové aplikace

Backendová část systému byla implementována v prostředí `Node.js` s využitím frameworku `Express`, který byl použit pro vytvoření REST API sloužícího ke komunikaci mezi klientskou částí aplikace a databází. Backend zajišťuje především správu dat systému, zpracování požadavků klienta a realizaci výpočtu návrhu rozmístění oddělení.

Struktura backendové aplikace byla navržena modulárním způsobem, aby byly jednotlivé části systému logicky odděleny podle své funkce. Tento přístup usnadňuje orientaci ve zdrojovém kódu, zvyšuje jeho přehlednost a umožňuje snadnější údržbu a rozšiřování systému.



Obrázek 4: Struktura projektu (vlastní zpracování)

Hlavní část implementace je umístěna ve složce src, která obsahuje jednotlivé moduly aplikace. Ty jsou rozděleny zejména do podsložek routes a utils. Ve složce routes jsou implementovány jednotlivé API endpointy systému, zatímco složka utils obsahuje pomocné moduly využívané v rámci backendové logiky aplikace.

Ve složce routes jsou implementovány samostatné moduly pro jednotlivé doménové entity systému. Každý z těchto modulů obsahuje definici příslušných REST API endpointů a zajišťuje zpracování HTTP požadavků souvisejících s danou entitou. V rámci systému byly vytvořeny zejména moduly pro správu budov, pater, oddělení, zaměstnanců a variant rozmístění oddělení.

Součástí implementace jsou také pomocné moduly umístěné ve složce utils. Tyto moduly obsahují zejména logiku pro práci s databází Neo4j a implementaci optimalizačního modelu využívaného pro výpočet rozmístění oddělení.

Vstupním bodem aplikace je soubor server.js, který zajišťuje spuštění HTTP serveru. Konfigurace samotné aplikace, registrace middleware a definice jednotlivých API rout jsou soustředěny v souboru app.js. Připojení k databázi Neo4j je realizováno prostřednictvím modulu db.js, který zajišťuje inicializaci databázového spojení a jeho využití v dalších částech aplikace.

Součástí je také složka tests, ve které jsou umístěny automatizované testy jednotlivých částí backendu. Tyto testy byly implementovány pomocí nástrojů `Jest` a Supertest a sloužily k ověřování správné funkčnosti API endpointů během vývoje systému.

4.3.2 Komunikace s databází Neo4j

Připojení k databázi bylo implementováno v samostatném modulu db.js, který zajišťuje inicializaci databázového driveru a jeho sdílení napříč jednotlivými částmi aplikace. Oddělení logiky připojení k databázi do samostatného modulu umožňuje centralizovanou správu databázového spojení a usnadňuje jeho využití v dalších částech backendu.

Konfigurační údaje potřebné pro připojení k databázi, jako je adresa databázového serveru, uživatelské jméno a heslo, byly ukládány do konfiguračního souboru `.env`. Tento přístup umožňuje oddělit citlivé údaje od zdrojového kódu aplikace a zvyšuje bezpečnost řešení.

```
1   const neo4j = require("neo4j-driver");
2   require("dotenv").config();
3
4   const driver = neo4j.driver(
5     process.env.NEO4J_URI,
6     neo4j.auth.basic(
7       process.env.NEO4J_USER,
8       process.env.NEO4J_PASSWORD
9     )
10  );
11
12  module.exports = { driver };
```

Obrázek 5: Inicializace připojení k databázi (vlastní zpracování)

Po inicializaci driveru bylo v jednotlivých částech aplikace vytvářeno databázové spojení pomocí tzv. session, které umožňuje provádět dotazy v jazyce Cypher. Databázové operace byly následně realizovány prostřednictvím těchto session, které byly po provedení dotazu uzavřeny.

Pro usnadnění práce s databází byl v projektu vytvořen také pomocný modul neo4jUtils.js, který obsahuje vybrané pomocné funkce pro práci s výsledky databázových dotazů. Tyto funkce zjednodušují zpracování vrácených dat a jejich převod do formátu využívaného v rámci REST API.

4.3.3 Implementace REST API

Komunikace mezi klientskou částí aplikace a backendem byla realizována prostřednictvím REST API, které bylo implementováno pomocí frameworku `Express`. API umožňuje

klientské aplikaci přistupovat k datům uloženým v databázi a provádět operace související se správou jednotlivých entit systému.

Jednotlivé API endpointy byly implementovány ve formě samostatných routerů, které jsou umístěny ve složce routes. Každý router je zodpovědný za zpracování požadavků souvisejících s konkrétní entitou systému. Tento přístup umožňuje přehledné oddělení jednotlivých částí API a usnadňuje další rozšiřování systému.

V rámci backendové aplikace byly implementovány následující moduly:

- buildingRoutes.js – správa budov
- floorRoutes.js – správa pater jednotlivých budov
- departmentRoutes.js – správa oddělení
- personRoutes.js – správa zaměstnanců
- assignmentRoutes.js – správa variant rozmístění oddělení
- mapRoutes.js – práce s mapou rozmístění oddělení

Každý z těchto modulů definuje jednotlivé endpointy, které zpracovávají HTTP požadavky typu GET, POST, PUT nebo DELETE. Tyto endpointy umožňují vytváření, úpravu, načítání a odstraňování dat v databázi.

Přehled hlavních endpointů implementovaného REST API je uveden v následující tabulce. Tyto endpointy umožňují aplikaci provádět operace nad jednotlivými entitami systému, jako je správa budov, oddělení, zaměstnanců nebo práce s mapou rozmístění.

Tabulka 1: Přehled endpointů (vlastní zpracování)

Endpoint	**Metoda**	**Popis**
`/health`	GET	health check API a ověření připojení k databázi
`/buildings`	GET	načtení seznamu budov
`/buildings`	POST	vytvoření nové budovy
`/buildings/:id`	PUT	úprava budovy
`/buildings/:id`	DELETE	odstranění budovy
`/buildings/:id/floors`	GET	načtení pater konkrétní budovy
`/buildings/:id/floors`	POST	vytvoření nového patra v dané budově
`/floors/:id`	PUT	úprava patra
`/floors/:id`	DELETE	odstranění patra
`/departments`	GET	načtení seznamu oddělení
`/departments`	POST	vytvoření nového oddělení
`/departments/:id/collaboration`	PUT	nastavení nebo změna vazby spolupráce oddělení
`/departments/:id`	PUT	úprava oddělení
`/departments/:id`	DELETE	odstranění oddělení
`/persons`	GET	načtení seznamu zaměstnanců
`/persons`	POST	vytvoření nového zaměstnance
`/persons/:id/department`	PUT	změna přiřazení zaměstnance k oddělení
`/persons/:id`	DELETE	odstranění zaměstnance
`/assignments/run`	POST	spuštění automatického výpočtu rozmístění oddělení
`/assignments/move`	POST	manuální přesun oddělení na cílové patro
`/assignments/:deptId/placement`	DELETE	odebrání oddělení z mapy rozmístění
`/assignments/clear`	POST	vytvoření nového prázdného snapshotu rozmístění
`/map`	GET	načtení celé aktuální mapy rozmístění
`/map/:buildingId`	GET	načtení mapy rozmístění pro konkrétní budovu

4.3.4 Implementace práce s mapou rozmístění

Součástí implementace backendové aplikace je také logika pro práci s mapou rozmístění oddělení v rámci jednotlivých budov a jejich pater. Tato část systému umožňuje zobrazovat aktuální rozmístění oddělení, provádět manuální úpravy jejich umístění a také spouštět automatický výpočet nového rozmístění.

Mapa rozmístění představuje aktuální stav přiřazení oddělení k jednotlivým patrům budov. Tento stav je v databázi reprezentován pomocí uzlů typu *Assignment* a *Placement*, které umožňují uchovávat jednotlivé varianty rozmístění oddělení.

Uzel *Assignment* reprezentuje konkrétní variantu rozmístění oddělení. Každý výpočet rozmístění vytváří novou instanci tohoto uzlu, čímž je možné uchovávat více variant rozmístění a mezi nimi přepínat. Uzel *Placement* pak reprezentuje konkrétní umístění oddělení na určité patro v rámci dané varianty rozmístění.

Manuální přesun oddělení

Uživatel může v rámci aplikace provádět manuální přesuny oddělení mezi jednotlivými patry. Tato operace je realizována prostřednictvím endpointu /assignments/move, který umožňuje změnit umístění vybraného oddělení na konkrétní patro.

Při provedení této operace je v databázi aktualizován odpovídající uzel *Placement*, který propojuje oddělení s cílovým patrem. Tento mechanismus umožňuje uživateli definovat pevné umístění vybraných oddělení ještě před spuštěním automatického výpočtu rozmístění.

Odebrání oddělení z mapy rozmístění

Systém také umožňuje odebrat oddělení z aktuální mapy rozmístění. Tato operace je realizována pomocí endpointu /assignments/{deptId}/placement, který odstraní existující vazbu mezi oddělením a patrem. Tímto způsobem lze oddělení z mapy rozmístění odebrat a následně jej znovu přiřadit buď manuálně, nebo prostřednictvím automatického výpočtu.

Vytvoření nového snapshotu rozmístění

Pro zahájení návrhu nové varianty rozmístění je možné vytvořit nový prázdný stav mapy rozmístění. Tato operace je realizována pomocí endpointu /assignments/clear, který vytvoří nový uzel Assignment reprezentující prázdnou variantu rozmístění. Díky tomu je možné zahájit návrh rozmístění zcela od začátku, aniž by došlo ke ztrátě předchozích variant uložených v databázi.

Zobrazení mapy rozmístění

Pro načtení aktuálního stavu rozmístění oddělení slouží endpoint /map, který vrací kompletní mapu rozmístění všech oddělení. Součástí API je také endpoint /map/{buildingId}, který umožňuje načíst mapu rozmístění pouze pro konkrétní budovu. Tato data jsou následně využívána pro vizualizaci rozmístění oddělení v rámci jednotlivých budov a jejich pater.

4.3.5 Implementace optimalizačního modelu

Nejdůležitější částí aplikace byl modul zajišťující automatický výpočet návrhu rozmístění oddělení do jednotlivých pater budov. Tato funkcionalita byla implementována v modulu ilpSolver.js, ve kterém byla úloha formulována jako problém celočíselného lineárního programování.

Cílem výpočtu bylo nalézt takové rozmístění oddělení, které respektuje kapacitní omezení jednotlivých pater, zohledňuje již pevně definovaná umístění vybraných oddělení a současně podporuje prostorovou blízkost spolupracujících oddělení. Optimalizační model byl řešen pomocí knihovny javascript-lp-solver.

Formulace proměnných modelu

Základ modelu tvořily binární proměnné reprezentující možné umístění oddělení na jednotlivá patra. Pro každou kombinaci volného oddělení a patra byla vytvořena proměnná **x_ij**, která nabývala hodnoty 1 v případě, že bylo oddělení přiřazeno na dané patro, a hodnoty 0 v opačném případě.

Vedle těchto proměnných byly zavedeny také binární proměnné **y_ikj**, které reprezentovaly situaci, kdy dvojice spolupracujících oddělení sdílí stejné patro. Tyto proměnné byly využity při konstrukci účelové funkce, protože umožňovaly zvýhodnit takové rozmístění, ve kterém jsou spolupracující oddělení umístěna co nejblíže.

Další skupinu tvořily pomocné proměnné **s_i**, které reprezentovaly stav, kdy určité oddělení nebylo umístěno na žádné patro. Tyto proměnné byly do modelu zahrnuty z důvodu penalizace nealokovaných oddělení.

Konstrukce účelové funkce

Účelová funkce byla navržena jako maximalizační. Jejím cílem bylo upřednostnit takové rozmístění, ve kterém spolupracující oddělení sdílejí stejné patro, a současně penalizovat situace, kdy některé oddělení zůstane nezařazeno.

Model tedy kombinoval dva principy. Prvním z nich byla odměna za splněnou spolupráci mezi odděleními. Pokud byla spolupracující dvojice umístěna na stejném patře, přispívala příslušná proměnná **y_ikj** kladně do hodnoty účelové funkce. Druhým principem byla penalizace za nealokovaná oddělení, která byla realizována pomocí pomocných proměnných **s_i**.

Tento přístup umožnil nalézt řešení, které se nesoustředilo pouze na splnění kapacitních omezení, ale současně zvýhodňovalo kvalitnější prostorové uspořádání z hlediska organizačních vazeb mezi odděleními.

Definice omezení modelu

V rámci modelu bylo definováno několik skupin omezení. První skupinu tvořilo omezení zajišťující, že každé volné oddělení může být umístěno nejvýše na jednom patře. Tím bylo zabráněno situaci, kdy by jedno oddělení bylo současně přiřazeno na více různých pater.

Druhou skupinu představovala kapacitní omezení jednotlivých pater. U každého patra byla zohledněna jeho maximální kapacita, přičemž od této hodnoty byla odečtena již obsazená kapacita způsobená odděleními s pevně definovaným umístěním. Teprve zbývající kapacita byla dále využita pro alokaci ostatních oddělení.

Další omezení byla spojena s pomocnými proměnnými **s_i**. Pro každé oddělení byla vytvořena rovnice, která vynucovala, že buď bude oddělení přiřazeno na jedno z dostupných pater, nebo bude označeno jako nealokované. Tím byl zajištěn jednoznačný vztah mezi alokačními proměnnými a penalizací za nezařazení oddělení.

Významnou část modelu tvořila také omezení související se spoluprací mezi odděleními. Pro dvojice volných spolupracujících oddělení byly definovány podmínky zajišťující, že proměnná **y_ikj** může nabýt hodnoty 1 pouze tehdy, pokud jsou obě oddělení skutečně umístěna na stejném patře.

```
69    for (const f of floors) {
70      // Kolik je patro už obsazeno zamknutými odděleními
71      const lockedOccupied = lockedPlacements
72        .filter(p => p.floorId === f.id)
73        .reduce((sum, p) => {
74          const d = departments.find(x => x.id === p.deptId);
75          return sum + (d ? d.size : 0);
76        }, 0);
77
78      const remaining = f.capacity - lockedOccupied;
79      const cName = `cap_${f.id}`;
80      model.constraints[cName] = { max: remaining };
81
82      for (const d of freeDepts) {
83        const name = xName(d.id, f.id);
84        if (!model.variables[name]) model.variables[name] = {};
85        model.variables[name][cName] = d.size;
86      }
87    }
```

Obrázek 6: Implementace kapacitního omezení pater (vlastní zpracování)

Na obrázku je ukázka kódu implementace kapacitního omezení jednotlivých pater. Pro každé patro je nejprve vypočtena již obsazená kapacita způsobená odděleními s pevně definovaným umístěním. Od celkové kapacity patra je následně odečtena tato již obsazená kapacita, čímž vznikne zbývající kapacita dostupná pro optimalizační model.

Následně je vytvořeno omezení, které zajišťuje, že součet velikostí oddělení přiřazených na dané patro nepřekročí dostupnou kapacitu. Každé potenciální přiřazení oddělení na patro přispívá do tohoto omezení hodnotou odpovídající velikosti oddělení.

Zohlednění pevně definovaných umístění a vzdálenosti oddělení

Model zohledňoval také již zamknutá umístění oddělení. Pokud byla obě spolupracující oddělení již pevně přiřazena, byl jejich přínos ke skóre spolupráce vyhodnocen přímo bez potřeby dalších optimalizačních proměnných. Pokud bylo zamknuto pouze jedno oddělení, byl bonus za spolupráci přičten přímo k proměnné reprezentující umístění druhého oddělení na stejné patro.

Současně byla do modelu zahrnuta omezení omezující maximální vzdálenost spolupracujících oddělení. Pokud se dvě patra nacházela v různých budovách nebo mezi nimi byl rozdíl větší než jedno podlaží, bylo zakázáno, aby na nich byla spolupracující oddělení současně umístěna. Obdobný princip byl aplikován také v situaci, kdy bylo jedno

oddělení již pevně zamknuto na konkrétním patře. V takovém případě bylo druhému oddělení zakázáno umístění na příliš vzdálená patra.

Tímto způsobem bylo dosaženo toho, že model nejen zvýhodňoval umístění spolupracujících oddělení na stejném patře, ale současně vylučoval řešení, ve kterých by byla tato oddělení prostorově příliš vzdálena.

Vyhodnocení výsledku

Po sestavení modelu byl spuštěn solver, který vrátil výsledné řešení ve formě hodnot jednotlivých proměnných. Na základě proměnných **x_ij**, jejichž výsledná hodnota byla rovna 1, bylo určeno konkrétní přiřazení oddělení k jednotlivým patrům.

Do výsledného seznamu byla následně doplněna také oddělení s pevně definovaným umístěním. Oddělení, pro která nebylo nalezeno žádné přípustné umístění, byla zařazena do seznamu nealokovaných oddělení. Současně bylo vypočteno výsledné skóre spolupráce, které zahrnovalo jak spolupráce splněné přímo mezi volně alokovanými odděleními, tak i případy, kdy byla spolupráce splněna díky již zamknutým umístěním.

Výsledné rozmístění bylo následně předáno další části backendu, kde bylo uloženo do databáze jako nová varianta rozmístění oddělení.

4.3.6 Implementace databázové vrstvy

Databázové schéma bylo vytvořeno pomocí dotazů v jazyce Cypher, které byly vykonány v prostředí Neo4j. Tyto dotazy definují zejména omezení unikátnosti identifikátorů jednotlivých entit. Pro účely reprodukovatelnosti projektu jsou tyto dotazy zároveň uloženy v samostatném souboru, který je součástí odevzdávaného zdrojového kódu.

```
3   CREATE CONSTRAINT building_id IF NOT EXISTS
4   FOR (b:Building) REQUIRE b.id IS UNIQUE;
5
6   CREATE CONSTRAINT floor_id IF NOT EXISTS
7   FOR (f:Floor) REQUIRE f.id IS UNIQUE;
8
9   CREATE CONSTRAINT department_id IF NOT EXISTS
10  FOR (d:Department) REQUIRE d.id IS UNIQUE;
11
12  CREATE CONSTRAINT person_id IF NOT EXISTS
13  FOR (p:Person) REQUIRE p.id IS UNIQUE;
14
15  CREATE CONSTRAINT assignment_id IF NOT EXISTS
16  FOR (a:Assignment) REQUIRE a.id IS UNIQUE;
17
18  CREATE CONSTRAINT placement_id IF NOT EXISTS
19  FOR (pl:Placement) REQUIRE pl.id IS UNIQUE;
```

Obrázek 7: Ukázka definice omezení v jazyce Cypher (vlastní zpracování)

Backendová aplikace komunikuje s databází Neo4j prostřednictvím oficiálního Neo4j JavaScript driveru. Připojení k databázi je konfigurováno pomocí environment proměnných, které obsahují adresu databázového serveru a přihlašovací údaje.

Inicializace připojení je realizována v samostatném modulu db.js, který vytváří instanci databázového driveru a zpřístupňuje ji ostatním částem aplikace.

```
1   const neo4j = require("neo4j-driver");
2   require("dotenv").config();
3
4   const driver = neo4j.driver(
5     process.env.NEO4J_URI,
6     neo4j.auth.basic(
7       process.env.NEO4J_USER,
8       process.env.NEO4J_PASSWORD
9     )
10  );
11
12  module.exports = { driver };
```

Obrázek 8: Vytvoření databázového připojen (vlastní zpracování)

4.4 Použité nástroje při vývoji a testování

Při vývoji aplikace bylo využito několik nástrojů podporujících implementaci, správu zdrojového kódu a testování jednotlivých částí systému. Tyto nástroje umožnily efektivní organizaci vývoje, průběžné ověřování funkčnosti implementovaných částí systému a usnadnily práci s jednotlivými komponentami backendové části aplikace.

4.4.1 Github

Pro správu verzí zdrojového kódu byl využit verzovací systém Git. Zdrojový kód projektu byl spravován prostřednictvím repozitáře na platformě GitHub, která umožňuje ukládání jednotlivých verzí projektu, sledování změn ve zdrojovém kódu a systematickou správu vývoje aplikace. Využití verzovacího systému umožnilo průběžně zaznamenávat jednotlivé úpravy implementace a zároveň poskytlo možnost návratu k předchozím verzím projektu v případě potřeby.

4.4.2 Postman

Pro testování implementovaného REST API byl využit nástroj Postman. V tomto nástroji byla vytvořena kolekce jednotlivých endpointů backendové aplikace, která umožňovala systematické testování funkcí systému během vývoje. Pomocí Postmanu bylo možné simulovat požadavky klientské aplikace, ověřovat správnost odpovědí serveru a kontrolovat chování jednotlivých API operací při různých vstupních datech. Tento přístup umožnil průběžně validovat správnou funkčnost backendové části systému ještě před napojením klientské aplikace.

4.4.3 Node.js a framework Express

Backendová část systému byla implementována v prostředí `Node.js`, které umožňuje vývoj serverových aplikací v jazyce JavaScript. Platforma `Node.js` je založena na asynchronním modelu zpracování požadavků a umožňuje efektivní práci s větším množstvím souběžných operací. Díky tomu je vhodná pro vývoj webových aplikací a REST API služeb.

Pro implementaci serverové logiky byl využit framework `Express`, který představuje minimalistický a rozšiřitelný nástroj pro vytváření webových aplikací v prostředí `Node.js`.

Tento framework poskytuje jednoduchý mechanismus pro definici jednotlivých API endpointů a zpracování HTTP požadavků. V rámci implementace backendu byly jednotlivé části systému rozděleny do samostatných routovacích modulů, které implementují operace nad jednotlivými entitami systému, například správu zaměstnanců, oddělení, budov nebo výpočet rozmístění oddělení. Takové rozdělení umožňuje přehlednou organizaci zdrojového kódu a usnadňuje další rozšiřování aplikace.

4.4.4 Neo4j AuraDB

Pro ukládání a správu dat byla využita grafová databáze Neo4j.V rámci implementace byla použita cloudová služba Neo4j AuraDB, která poskytuje spravovanou databázovou instanci dostupnou prostřednictvím vzdáleného připojení. Využití této služby umožnilo provoz databáze bez nutnosti lokální instalace databázového serveru a zároveň poskytlo stabilní prostředí pro vývoj i testování aplikace. Připojení k databázi bylo realizováno pomocí oficiálního JavaScriptového driveru Neo4j, který umožňuje komunikaci s databází prostřednictvím dotazovacího jazyka Cypher.

4.4.5 Jest a Supertest

Pro automatizované testování backendové části aplikace byly využity nástroje `Jest` a Supertest. Tyto nástroje byly použity k ověřování správné funkčnosti REST API implementovaného pomocí frameworku `Express`.

V rámci projektu byly vytvořeny testy, které simulují HTTP požadavky na jednotlivé API endpointy a ověřují správnost odpovědí serveru. Testy se zaměřují zejména na ověření základní funkčnosti jednotlivých částí systému, například správy budov, oddělení nebo zaměstnanců.

Automatizované testy byly využívány především během vývoje backendu k průběžné kontrole funkčnosti implementovaných částí systému a k ověřování, že změny v kódu nevedly k narušení již implementované funkcionality.

5 Výsledky a diskuse

5.1 Vyhodnocení navrženého řešení

Na základě realizované implementace lze konstatovat, že navržený systém splňuje definované funkční požadavky stanovené v kapitole 4.1.2. Byla implementována podpora pro správu všech klíčových entit systému, tedy budov, pater, oddělení a zaměstnanců, včetně jejich vzájemných vztahů.

Systém umožňuje evidenci kapacit jednotlivých pater, určování velikosti oddělení na základě počtu zaměstnanců a definování vazeb spolupráce mezi odděleními. Tyto vazby představují zásadní vstup pro optimalizační model, který na jejich základě upřednostňuje prostorovou blízkost spolupracujících oddělení.

Součástí implementace je rovněž práce s mapou rozmístění oddělení, která je reprezentována pomocí uzlů typu *Assignment* a *Placement*. Tento přístup umožňuje uchovávat více variant rozmístění a mezi nimi přepínat.

Systém podporuje jak automatické generování návrhu rozmístění prostřednictvím optimalizačního modelu, tak i manuální zásahy uživatele, například přesun oddělení na konkrétní patro nebo vytvoření nové varianty rozmístění.

5.1.1 Vyhodnocení optimalizačního modelu

Nejdůležitější částí systému je optimalizační model formulovaný jako úloha celočíselného lineárního programování. Model pracuje s binárními proměnnými reprezentujícími přiřazení oddělení k jednotlivým patrům a s proměnnými vyjadřujícími společné umístění spolupracujících oddělení.

Cílem modelu je nalézt takové rozmístění oddělení, které respektuje kapacitní omezení jednotlivých pater, zohledňuje pevně definovaná umístění, maximalizuje míru splnění vazeb spolupráce mezi odděleními a minimalizuje vzdálenost mezi spolupracujícími odděleními.

Výsledky ukazují, že model efektivně preferuje umístění spolupracujících oddělení na stejném patře. V případech, kdy to kapacitní omezení neumožňují, dochází k jejich umístění na patra bezprostředně sousedící. Zároveň model vylučuje řešení, ve kterých by byla spolupracující oddělení umístěna na výrazně vzdálených patrech.

Po vyřešení optimalizační úlohy jsou výsledky interpretovány na základě hodnot proměnných, ze kterých je odvozeno konkrétní přiřazení oddělení k patrům. Oddělení, pro

která není nalezeno přípustné umístění, jsou evidována jako nealokovaná, což umožňuje uživateli identifikovat problematická místa v návrhu.

5.1.2 Kombinace automatického a manuálního přístupu

Významnou vlastností systému je možnost kombinace automatického výpočtu a manuálních zásahů uživatele. Uživatel může před spuštěním optimalizačního modelu definovat pevné umístění vybraných oddělení, které je následně při výpočtu respektováno.

Tento přístup umožňuje zohlednit specifické požadavky, které nelze jednoduše formalizovat v rámci matematického modelu. Typickým příkladem může být umístění vedení společnosti nebo oddělení se specifickými provozními požadavky.

Manuální operace, jako je přesun oddělení mezi patry nebo odebrání oddělení z mapy rozmístění, jsou realizovány prostřednictvím REST API a okamžitě se promítají do aktuální varianty rozmístění.

5.1.3 Přínosy využití grafové databáze

Použití grafové databáze Neo4j se ukázalo jako vhodné zejména z hlediska modelování vztahů mezi entitami systému. Vazby mezi odděleními, které vyjadřují jejich spolupráci, jsou v grafovém modelu reprezentovány přirozeným způsobem pomocí hran mezi uzly.

Tento přístup umožňuje efektivní práci s propojenými daty bez nutnosti složitých spojení, která jsou typická pro relační databáze. Grafový model zároveň usnadňuje implementaci dotazů zaměřených na analýzu vztahů mezi odděleními.

Další výhodou je schopnost přirozeně pracovat i s neúplnými daty. Například patra bez přiřazených oddělení jsou v grafové databázi zahrnuta automaticky, zatímco v relačním modelu by bylo nutné použít speciální typy spojení, například LEFT JOIN.

Grafová databáze zároveň poskytuje vysokou flexibilitu datového modelu, což umožňuje snadné rozšíření systému o další typy vztahů nebo atributů bez nutnosti zásadních změn struktury databáze.

5.1.4 Celkové zhodnocení

Navržené řešení představuje funkční nástroj pro podporu rozhodování při rozmísťování oddělení v rámci organizace. Kombinace grafového datového modelu a optimalizačního přístupu umožňuje efektivně řešit problém, který je v praxi často realizován manuálně.

Systém je schopen generovat smysluplné návrhy rozmístění, které respektují definovaná omezení a zároveň reflektují vztahy mezi odděleními. Díky možnosti kombinace automatického výpočtu a manuálních zásahů je řešení dostatečně flexibilní pro praktické využití.

5.2 Ověření řešení pomocí testovacího scénáře

5.2.1 Popis testovacího scénáře

Pro ověření funkčnosti navrženého řešení byl vytvořen modelový scénář simulující situaci středně velké společnosti, která plánuje rozmístění svých oddělení do dostupných budov a jejich pater.

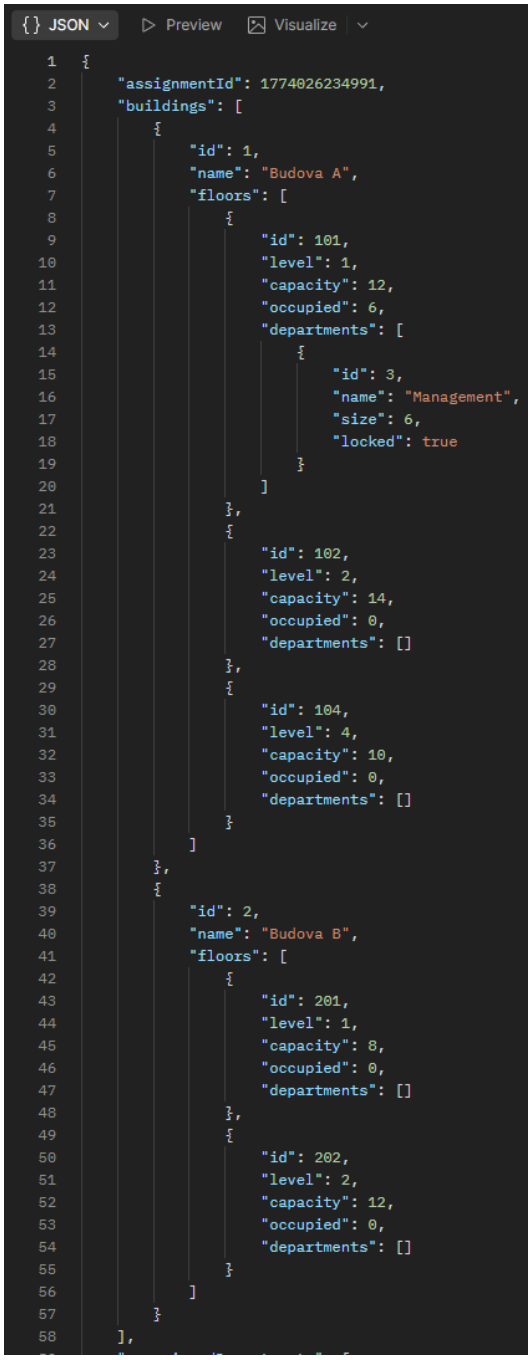
V rámci scénáře byly definovány dvě budovy s několika patry, přičemž každé patro mělo určenu maximální kapacitu vyjádřenou počtem zaměstnanců. Do systému bylo zadáno šest oddělení o různých velikostech, přičemž velikost oddělení byla odvozena od počtu zaměstnanců přiřazených k danému oddělení.

Mezi vybranými odděleními byly definovány vazby spolupráce reprezentující intenzivní pracovní interakce. Konkrétně se jednalo o vazby mezi odděleními HR a Management a dále Marketing a Obchod. Tyto vazby slouží jako vstup pro optimalizační model, který se snaží minimalizovat vzdálenost mezi spolupracujícími odděleními.

Součástí scénáře bylo také definování pevného umístění pro vybrané oddělení. Oddělení Marketing bylo manuálně umístěno na první patro budovy A ještě před spuštěním optimalizačního výpočtu. Tímto způsobem bylo simulováno reálné omezení, kdy je nutné zachovat konkrétní umístění vybraných organizačních jednotek.

5.2.2 Průběh výpočtu rozmístění

Po inicializaci vstupních dat byl vytvořen nový prázdný stav rozmístění prostřednictvím příslušného API endpointu. Následně bylo provedeno manuální umístění oddělení Marketing na první patro budovy A.

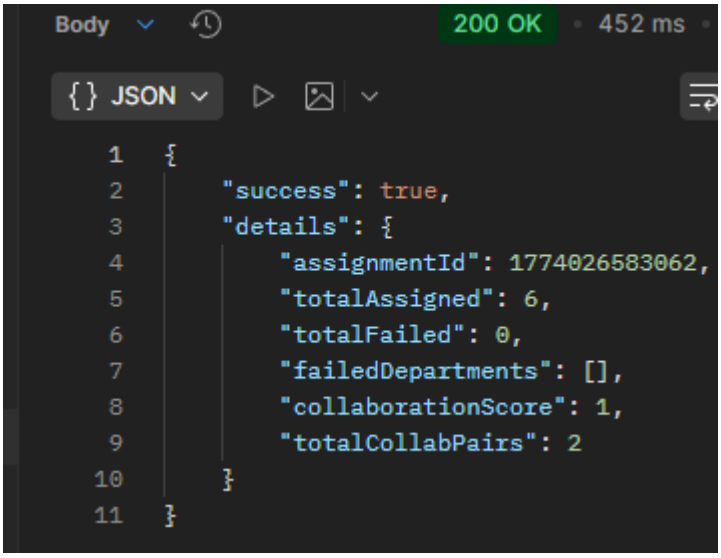


Obrázek 9: Ukázka stavu po manuální umístění oddělení Marketing (vlastní zpracování)

Po nastavení vstupních podmínek byl spuštěn optimalizační model, který na základě definovaných parametrů sestavil úlohu celočíselného lineárního programování a následně ji vyřešil pomocí implementovaného solveru.

Během výpočtu byla zohledněna kapacitní omezení jednotlivých pater Dále bylo zohledněno pevně definované umístění oddělení Marketing a nakonec i vazby spolupráce mezi odděleními.

Výsledkem výpočtu bylo přiřazení jednotlivých oddělení ke konkrétním patrům. Toto přiřazení bylo následně uloženo do databáze jako nová varianta rozmístění reprezentovaná uzlem typu *Assignment*.



Obrázek 10: Spuštění automatického usazení (vlastní zpracování)

5.2.3 Vyhodnocení výsledného rozmístění

Výsledné rozmístění oddělení bylo analyzováno z hlediska splnění definovaných požadavků. Z výsledků vyplývá, že oddělení Management a HR byla umístěna na první patro budovy A. Celková obsazenost tohoto patra dosáhla hodnoty 11 z maximální kapacity 12, což znamená efektivní využití dostupného prostoru při zachování rezervy pro případné další umístění.

Na druhém patře budovy A byla umístěna oddělení Finance a Marketing. I zde došlo k efektivnímu využití kapacity, kdy obsazenost činila 13 z maximálních 14 míst.

Oddělení Obchod bylo umístěno na třetí patro budovy A, kde obsadilo 9 z 10 dostupných míst. Přestože nebylo umístěno na stejném patře jako oddělení Marketing, zůstalo ve stejné budově, čímž byla zachována alespoň částečná prostorová blízkost mezi těmito spolupracujícími odděleními.

První patro budovy B zůstalo neobsazené, což poukazuje na skutečnost, že optimalizační model preferoval soustředění oddělení do jedné budovy s cílem minimalizovat

vzdálenosti mezi nimi. Výjimkou bylo pouze IT oddělení, které se již do první budovy nevešlo a bylo umístěno na druhé patro budovy B, kde obsadilo 10 z 12 dostupných míst. Toto umístění reflektuje kapacitní omezení ostatních pater.

Důležitým zjištěním je, že v rámci výsledného řešení nedošlo k překročení kapacitních omezení žádného patra a všechna oddělení byla úspěšně alokována, což potvrzuje i prázdný seznam nealokovaných oddělení.

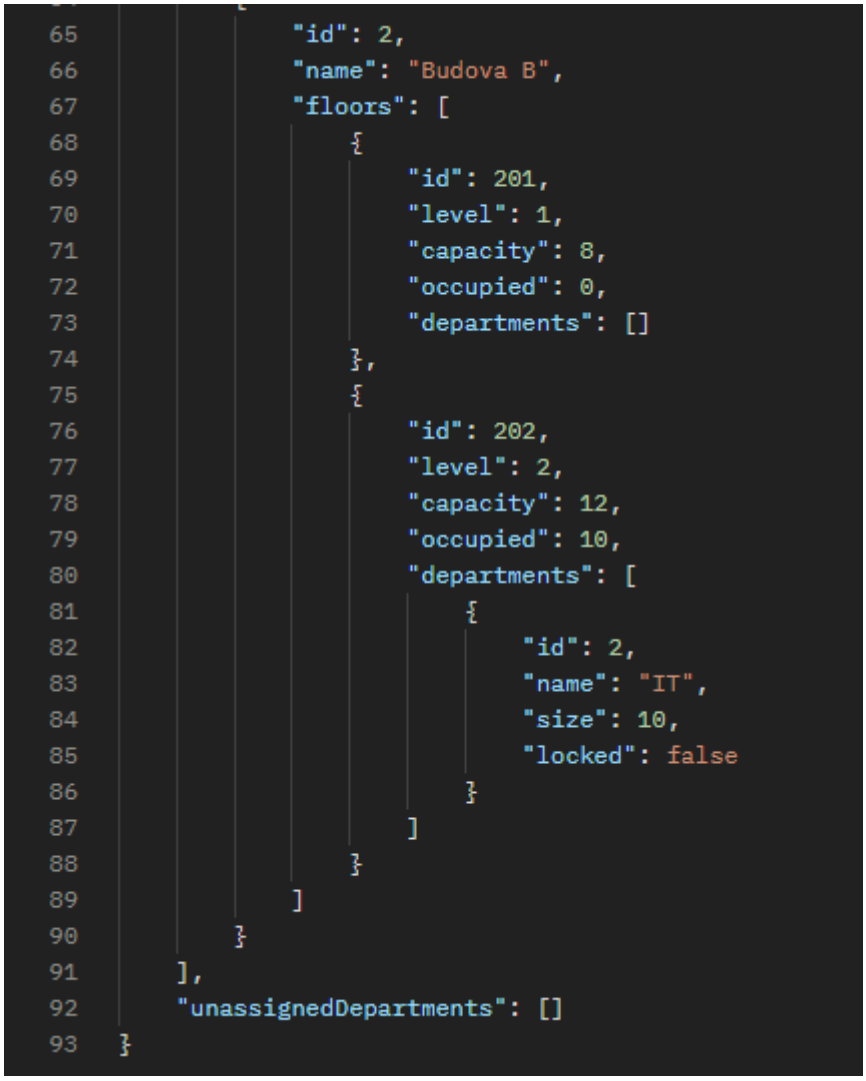
Z hlediska vazeb spolupráce lze konstatovat, že oddělení HR a Management byla umístěna na stejném patře, což odpovídá požadavku na jejich úzkou spolupráci. Dále. Že oddělení Marketing a Obchod byla umístěna ve stejné budově, i když na různých patrech, což představuje kompromisní řešení vzhledem ke kapacitním omezením.

Na základě těchto výsledků lze konstatovat, že optimalizační model dokázal nalézt řešení, které efektivně vyvažuje jednotlivé požadavky a omezení. Výsledné rozmístění respektuje kapacity a současně zohledňuje definované vazby spolupráce

Pro lepší přehlednost výsledného rozmístění oddělení jsou níže uvedeny obrázky znázorňující finální stav rozmístění v jednotlivých budovách a jejich patrech.

```
{
    "assignmentId": 1774026583062,
    "buildings": [
        {
            "id": 1,
            "name": "Budova A",
            "floors": [
                {
                    "id": 101,
                    "level": 1,
                    "capacity": 12,
                    "occupied": 11,
                    "departments": [
                        {
                            "id": 3,
                            "name": "Management",
                            "size": 6,
                            "locked": true
                        },
                        {
                            "id": 4,
                            "name": "HR",
                            "size": 5,
                            "locked": false
                        }
                    ]
                },
                {
                    "id": 102,
                    "level": 2,
                    "capacity": 14,
                    "occupied": 13,
                    "departments": [
                        {
                            "id": 5,
                            "name": "Finance",
                            "size": 7,
                            "locked": false
                        },
                        {
                            "id": 1,
                            "name": "Marketing",
                            "size": 6,
                            "locked": false
                        }
                    ]
                },
                {
                    "id": 103,
                    "level": 3,
                    "capacity": 10,
                    "occupied": 9,
                    "departments": [
                        {
                            "id": 6,
                            "name": "Obchod",
                            "size": 9,
                            "locked": false
                        }
```


Obrázek 11: Výsledek automatického usazení 1/2 (vlastní zpracování)



Obrázek 12: Výsledek automatického usazení 2/2 (vlastní zpracování)

5.3 Možnosti dalšího rozvoje systému

Navržené řešení pro rozmístění oddělení do budov a jejich pater představuje funkční nástroj pro podporu rozhodování, nicméně je nutné zohlednit i jeho omezení a zároveň identifikovat možnosti dalšího rozvoje.

Jedním z omezení je zjednodušený model prostorových vzdáleností. V současné implementaci je vzdálenost mezi odděleními vyjádřena pouze na úrovni pater a budov, přičemž model zvýhodňuje umístění na stejném nebo sousedním patře. Tento přístup neodpovídá plně reálnému uspořádání pracovního prostředí. Možným rozšířením je proto zavedení přesnějšího modelu vzdáleností, například na základě skutečné dispozice budov nebo času potřebného k přesunu mezi pracovišti.

Model dále pracuje s omezeným množstvím vstupních faktorů. Zohledňuje především kapacity pater, velikost oddělení a vazby spolupráce mezi nimi, avšak nezahrnuje další aspekty, které mohou v praxi ovlivňovat rozmístění oddělení. Mezi tyto faktory patří například požadavky na typ pracovního prostředí, potřeba oddělení určitých týmů, technologické vybavení pracovišť nebo bezpečnostní omezení. Rozšíření modelu o tyto prvky by vedlo k realističtějším výsledkům.

Významným faktorem je také kvalita vstupních dat. Přesnost výsledného návrhu je přímo závislá na správnosti definovaných kapacit, velikostí oddělení a zejména vazeb spolupráce. V případě nepřesných nebo neúplných dat může model generovat řešení, která neodpovídají skutečným potřebám organizace. Do budoucna by proto bylo vhodné zajistit automatizovaný sběr dat, například integrací s podnikovými systémy.

Další možností rozvoje je rozšíření analytických schopností systému. Například by bylo možné umožnit porovnání více variant rozmístění nebo zavést metriky pro hodnocení kvality jednotlivých řešení. To by uživatelům poskytlo lepší podporu při výběru nejvhodnější varianty.

6 Závěr

Cílem této diplomové práce bylo navrhnout a implementovat backendové řešení pro podporu rozmístění oddělení do budov a pater s využitím grafové databáze Neo4j a optimalizačního modelu.

Tento cíl byl splněn. Byla navržena a implementována backendová aplikace, která umožňuje evidenci všech relevantních entit systému, správu jejich vztahů a automatický výpočet návrhu rozmístění oddělení. Systém využívá grafovou databázi pro efektivní modelování vztahů a optimalizační model založený na celočíselném lineárním programování pro výpočet výsledného řešení.

Výsledkem práce je funkční prototyp systému, který nahrazuje manuální přístup k řešení problému systematickým a automatizovaným procesem. Systém umožňuje generovat návrhy rozmístění, které respektují kapacitní omezení a zároveň zohledňují spolupráci mezi odděleními.

Práce zároveň prokázala, že grafové databáze představují vhodný nástroj pro řešení problémů, kde hrají klíčovou roli vztahy mezi entitami. Ve srovnání s relačním přístupem umožňují přirozenější modelování dat a jednodušší práci s vazbami.

7 Seznam použitých zdrojů

1. LAUDON, Kenneth C.; LAUDON, Jane P. Management Information Systems: Managing the Digital Firm. 16th ed. Pearson, 2020.ISBN 9781292296562.
2. TECHTARGET. Decision suport system (DSS). Online. 2025. Dostupné z: https://www.techtarget.com/searchcio/definition/decision-support-system. [cit. 2026-03-14].
3. IBM. What is a relational database. Online. 2022. Dostupné z: https://www.ibm.com/think/topics/data-modeling. [cit. 2026-03-14].
4. IBM. What is a relational database. Online. 2022. Dostupné z: https://www.ibm.com/think/topics/relational-databases. [cit. 2026-03-14].
5. ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. Graph Databases. 2nd ed. O'Reilly Media, 2015. ISBN 978-1-4919-3213-6.
6. Van BRUGGEN, Rik, Learning Neo4j. Published by Packt Publishing Ltd. Livery Place 35 Livery Street Birmingham B3 2PB, UK. 2025, ISBN 978-1-84951-716-4
7. LYON, William. Full stack GraphQL applications : with React, Node.js, and Neo4j. Shelter Island, New York: Manning Publication Co., 2022. ISBN 9781617297038.
8. GEEKSFORGEEKS. Client-Server Model. Online. 2026. Dostupné z: https://www.geeksforgeeks.org/system-design/client-server-model/. [cit. 2026-03-14].
9. FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures [online]. Dissertation. University of California, Irvine, 2000. Dostupné z: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf [cit. 2026-03-14].
10. MDN WEB DOCS. HTTP request methods. Online. 2025. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods. [cit. 2026-03-14].
11. MDN WEB DOCS. Server-side. Online. 2025. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side. [cit. 2026-03-14].
12. MDN WEB DOCS. JavaScript. Online. 2025. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript. [cit. 2026-03-14].
13. NODE.JS. About Node.js. Online. 2023. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript. [cit. 2026-03-14].
14. MDN WEB DOCS. Express/Node introduction. Online. 2025. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/Introduction. [cit. 2026-03-15].
15. TAHA, Hamdy A. Operations Research: An Introduction. 10th ed. Pearson, 2017. ISBN 978-0134444017.
16. ENCYCLOPEADIA BRITANNICA. Optimization. Online. 2023. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/Introduction. [cit. 2026-03-15].
17. MASSACHUSETTS INSTITUTE OF TECHNOLOGY. Integer Programming. Online. 2010. Dostupné z: https://web.mit.edu/15.053/www/AMP-Chapter-09.pdf. [cit. 2026-03-15].

18. IBM. An introduction to UML. Online. 2023. Dostupné z: https://developer.ibm.com/articles/an-introduction-to-uml/. [cit. 2026-03-15].
19. LUCID SOFTWARE. UML Yse Case Duafran Tutorial. Online. 2026. Dostupné z: https://www.lucidchart.com/pages/tutorial/uml-use-case-diagram. [cit. 2026-03-15].
20. IBM. The class diagram. Online. 2023. Dostupné z: https://developer.ibm.com/articles/the-class-diagram/. [cit. 2026-03-15].
21. MYERS, Glenford J., SANDLER, Corey a BADGETT, Tom. The Art of Software Testing. 3rd ed. Hoboken: John Wiley & Sons, 2011. ISBN 978-1118031964.
22. GEEKSFORGEEKS. API Testing in Software Testing. Online. 2026. Dostupné z: https://www.geeksforgeeks.org/software-engineering/api-testing-software-testing/. [cit. 2026-03-15].
23. GEEKSFORGEEKS. Introduction to Postman for API Development. Online. 2025. Dostupné z: https://www.geeksforgeeks.org/web-tech/introduction-postman-api-development/. [cit. 2026-03-15].
24. META PLATFORMS, INC. Jest Documentation. Online. 2025. Dostupné z: https://jestjs.io/docs/getting-started. [cit. 2026-03-15].
25. LADJS. Supertest. Online. 2024. Dostupné z: https://github.com/forwardemail/supertest. [cit. 2026-03-15].
26. CHACON, Scott a STRAUB, Ben. Pro Git. 2nd ed. Berkeley: Apress, 2014. ISBN 978-1484200773.
27. COURSERA. What is GitHub? Online. 2025. Dostupné z: https://www.coursera.org/articles/what-is-github. [cit. 2026-03-15].
28. LEBEDA, Štěpán. OfficePlanner. Online. 2026. Dostupné z: https://github.com/GrandpaLebeda/office-planner

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

Obrázek 1: Client – server model 22
Obrázek 2: UML diagram tříd datového modelu systému 46
Obrázek 3: Diagram případů užití 49
Obrázek 5: Struktura projektu 50
Obrázek 6: Inicializace připojení k databázi 52
Obrázek 7: Implementace kapacitního omezení pater 58
Obrázek 8: Ukázka definice omezení v jazyce Cypher 60
Obrázek 9: Vytvoření databázového připojen 60
Obrázek 10: Ukázka stavu po manuální umístění oddělení Marketing 66
Obrázek 11: Spuštění automatického usazení 67
Obrázek 12: Výsledek automatického usazení 1/2 69
Obrázek 13: Výsledek automatického usazení 2/2 70

8.2 Seznam tabulek

Tabulka 1: Přehled endpointů 54

8.3 Seznam použitých zkratek

API – Application Programming Interface

CRUD – Create, Read, Update, Delete

DSS – Decision Support System / systém pro podporu rozhodování

FP – Funkční požadavek

HR – Human Resources

HTTP – Hypertext Transfer Protocol

I/O – Input/Output

IT – Information Technology

JSON – JavaScript Object Notation

NP – Nefunkční požadavek

npm – Node Package Manager

REST – Representational State Transfer

SQL – Structured Query Language

UML – Unified Modeling Language

URI – Uniform Resource Identifier

Přílohy

Součástí příloh této diplomové práce je kompletní zdrojový kód implementované aplikace. Pro snadnější přístup a případné další využití je zdrojový kód rovněž dostupný prostřednictvím repozitáře na platformě GitHub (Lebeda, 2026). Repozitář obsahuje backendovou i frontendovou část aplikace, přičemž předmětem této práce je backendová část, která je umístěna ve složce *backend*, spolu s doprovodnou dokumentací ve složce *docs*.

Obsah příloh:

assignmentRoutes.js **77**

buildingRoutes.js **82**

departmentRoutes.js **86**

floorRoutes.js **90**

mapRoutes.js **92**

personRoutes.js **95**

ilpSolver.js **99**

neo4jUtils.js **103**

app.js **103**

db.js **104**

server.js **104**

assignments.test.js **105**

buildings.test.js **107**

departments.test.js **109**

health.test.js **111**

testHelpers.js **113**

Databáze **114**

assignmentRoutes.js

```
const express = require("express");
const { driver } = require("../db");
const { toNum } = require("../utils/neo4jUtils");
const { solveAllocation } = require("../utils/ilpSolver");

const router = express.Router();

// POST /assignment/run
router.post("/run", async (req, res) => {
  const session = driver.session();
  try {
    const deptRes = await session.run(`
      MATCH (d:Department)
      OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
      RETURN d.id AS id, d.name AS name, count(p) AS size
    `);
    let departments = deptRes.records.map(r => ({
      id: toNum(r.get("id")),
      name: r.get("name"),
      size: toNum(r.get("size"))
    }));

    const floorRes = await session.run(`
      MATCH (b:Building)-[:HAS_FLOOR]->(f:Floor)
      RETURN f.id AS id, f.capacity AS capacity, f.level AS level, b.name AS buildingName
      ORDER BY b.id, f.level
    `);
    let floors = floorRes.records.map(r => ({
      id: toNum(r.get("id")),
      capacity: toNum(r.get("capacity")),
      buildingName: r.get("buildingName"),
      occupied: 0,
      assignedDepts: []
    }));

    const lastARes = await session.run(`
      MATCH (a:Assignment)
      RETURN a.id AS id
      ORDER BY a.createdAt DESC
      LIMIT 1
    `);

    let lockedPlacements = [];
    if (lastARes.records.length > 0) {
      const lastId = toNum(lastARes.records[0].get("id"));
      const lockedRes = await session.run(`
        MATCH (a:Assignment {id: $lastId})-[:HAS_PLACEMENT]->(pl:Placement {locked: true})
        MATCH (pl)-[:OF_DEPARTMENT]->(d:Department)
        MATCH (pl)-[:ON_FLOOR]->(f:Floor)
        RETURN d.id AS deptId, f.id AS floorId
      `, { lastId });
      lockedPlacements = lockedRes.records.map(r => ({
        deptId: toNum(r.get("deptId")),
        floorId: toNum(r.get("floorId"))
      }));
    }

    const lockedDeptIds = new Set(lockedPlacements.map(p => p.deptId));
    lockedPlacements.forEach(lp => {
```

```
      const d = departments.find(x => x.id === lp.deptId);
      const f = floors.find(x => x.id === lp.floorId);
      if (d && f) {
        f.occupied += d.size;
        f.assignedDepts.push({ ...d, locked: true });
      }
    });

    const collabRes = await session.run(`
      MATCH (d:Department)-[:COLLABORATES_WITH]->(p:Department)
      RETURN d.id AS deptId, p.id AS partnerId
    `);
    const collaborations = collabRes.records.map(r => ({
      deptId: toNum(r.get("deptId")),
      partnerId: toNum(r.get("partnerId"))
    }));

    if (departments.length === 0) {
      return res.status(400).json({ error: "Žádná oddělení k alokaci." });
    }
    if (floors.length === 0) {
      return res.status(400).json({ error: "Žádná dostupná patra k alokaci." });
    }
    const totalCapacity = floors.reduce((sum, f) => sum + f.capacity, 0);
    if (totalCapacity === 0) {
      return res.status(400).json({ error: "Celková kapacita všech dostupných pater je 0."
});
    }

    const { assignments, failed, collaborationScore, totalCollabPairs } = solveAllocation({
      departments,
      floors,
      lockedPlacements,
      collaborations,
    });

    const newAssignmentId = Date.now();
    await session.run(`CREATE (a:Assignment {id: $id, createdAt: datetime()})`, { id:
newAssignmentId });

    for (const asgn of assignments) {
      await session.run(`
        MATCH (a:Assignment {id:$aId}), (dept:Department {id:$dId}), (fl:Floor {id:$fId})
        CREATE (a)-[:HAS_PLACEMENT]->(pl:Placement {
          id: "PL_"+$aId+"_"+$dId,
          locked: $l,
          updatedAt: datetime(),
          source: "automatic_ilp"
        }),
        (pl)-[:OF_DEPARTMENT]->(dept),
        (pl)-[:ON_FLOOR]->(fl)
      `, { aId: newAssignmentId, dId: asgn.deptId, fId: asgn.floorId, l: asgn.locked });
    }

    res.json({
      success: true,
      details: {
        assignmentId: newAssignmentId,
        totalAssigned: departments.length - failed.length,
        totalFailed: failed.length,
        failedDepartments: failed,
        collaborationScore,
        totalCollabPairs,
```

```
      }
    });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// POST /assignments/move
router.post("/move", async (req, res) => {
  const session = driver.session();
  const { departmentId, targetFloorId } = req.body;
  try {
    const dataRes = await session.run(`
      MATCH (d:Department {id: $deptId})
      OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
      WITH d, count(p) AS deptSize
      MATCH (f:Floor {id: $floorId})
      OPTIONAL MATCH (lastA:Assignment)
      WITH d, deptSize, f, lastA ORDER BY lastA.createdAt DESC LIMIT 1
      /* Výpočet obsazenosti cílového patra v posledním stavu */
      OPTIONAL MATCH (lastA)-[:HAS_PLACEMENT]->(pl:Placement)-[:ON_FLOOR]->(f)
      OPTIONAL MATCH (pl)-[:OF_DEPARTMENT]->(otherD:Department)
      OPTIONAL MATCH (otherD)<-[:WORKS_IN]-(otherP:Person)
      RETURN d.id AS deptId, deptSize, f.id AS floorId, f.capacity AS capacity,
             count(otherP) AS currentOccupied, lastA.id AS lastAssignmentId
    `, { deptId: Number(departmentId), floorId: Number(targetFloorId) });

    if (dataRes.records.length === 0) return res.status(404).json({ error: "Oddělení nebo
patro nebylo nalezeno." });

    const r = dataRes.records[0];
    const dSize = toNum(r.get("deptSize"));
    const cap = toNum(r.get("capacity"));
    const occ = toNum(r.get("currentOccupied"));
    const lastAId = toNum(r.get("lastAssignmentId"));

    if (occ + dSize > cap) {
      return res.status(400).json({
        error: "Nedostatečná kapacita patra.",
        details: { capacity: cap, needed: occ + dSize, currentOccupied: occ }
      });
    }

    const newId = Date.now();
    await session.run(`CREATE (a:Assignment {id: $id, createdAt: datetime()})`, { id: newId
});

    if (lastAId) {
      await session.run(`
        MATCH    (oldA:Assignment    {id:    $oldId})-[:HAS_PLACEMENT]->(oldPl:Placement)-
[:OF_DEPARTMENT]->(d:Department)
        WHERE d.id <> $movingDeptId
        MATCH (oldPl)-[:ON_FLOOR]->(f:Floor), (newA:Assignment {id: $newId})
        CREATE (newA)-[:HAS_PLACEMENT]->(newPl:Placement {
          id: "PL_"+$newId+"_"+d.id,
          locked: oldPl.locked,
          updatedAt: datetime(),
          source: "cloned"
        }),
        (newPl)-[:OF_DEPARTMENT]->(d),
        (newPl)-[:ON_FLOOR]->(f)
```

```
      `, { oldId: lastAId, newId: newId, movingDeptId: Number(departmentId) });
    }

    await session.run(`
      MATCH (newA:Assignment {id: $newId}), (d:Department {id: $deptId}), (f:Floor {id:
$fId})
      CREATE (newA)-[:HAS_PLACEMENT]->(pl:Placement {
        id: "PL_"+$newId+"_"+$deptId,
        locked: true,
        updatedAt: datetime(),
        source: "manual"
      }),
      (pl)-[:OF_DEPARTMENT]->(d),
      (pl)-[:ON_FLOOR]->(f)
    `, { newId: newId, deptId: Number(departmentId), fId: Number(targetFloorId) });

    res.json({ success: true, assignmentId: newId, message: "Manuální přesun úspěšný." });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// DELETE /assignments/:deptId/placement
router.delete("/:deptId/placement", async (req, res) => {
  const session = driver.session();
  const removingDeptId = Number(req.params.deptId);

  if (isNaN(removingDeptId)) return res.status(400).json({ error: "ID oddělení musí být
číslo." });

  try {
    const lastARes = await session.run(`
      MATCH (a:Assignment)
      RETURN a.id AS id ORDER BY a.createdAt DESC LIMIT 1
    `);

    if (lastARes.records.length === 0) {
      return res.status(404).json({ error: "Žádná aktivní alokace nenalezena." });
    }

    const lastAId = toNum(lastARes.records[0].get("id"));
    const newId = Date.now();

    await session.run(`CREATE (a:Assignment {id: $id, createdAt: datetime()})`, { id: newId
});

    await session.run(`
      MATCH     (oldA:Assignment     {id:     $oldId})-[:HAS_PLACEMENT]->(oldPl:Placement)-
[:OF_DEPARTMENT]->(d:Department)
      WHERE d.id <> $removingDeptId
      MATCH (oldPl)-[:ON_FLOOR]->(f:Floor), (newA:Assignment {id: $newId})
      CREATE (newA)-[:HAS_PLACEMENT]->(newPl:Placement {
        id: "PL_"+$newId+"_"+d.id,
        locked: oldPl.locked,
        updatedAt: datetime(),
        source: "cloned_removal"
      }),
      (newPl)-[:OF_DEPARTMENT]->(d),
      (newPl)-[:ON_FLOOR]->(f)
    `, { oldId: lastAId, newId: newId, removingDeptId: removingDeptId });
```

```
    res.json({ success: true, assignmentId: newId, message: "Odebrání týmu z mapy úspěšné."
});
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// POST /assignments/clear
router.post("/clear", async (req, res) => {
  const session = driver.session();
  const newId = Date.now();
  try {
    await session.run(`CREATE (a:Assignment {id: $id, createdAt: datetime()})`, { id: newId
});
    res.json({ success: true, assignmentId: newId, message: "Mapa vyčištěna." });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

module.exports = router;
```

buildingRoutes.js

```
const express = require("express");
const { driver } = require("../db");
const { toNum } = require("../utils/neo4jUtils");

const router = express.Router();

// GET /buildings
router.get("/", async (req, res) => {
  const session = driver.session();
  try {
    const result = await session.run(`MATCH (b:Building) RETURN b.id AS id, b.name AS name
ORDER BY id`);
    res.json(result.records.map(r => ({ id: toNum(r.get("id")), name: r.get("name") })));
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// POST /buildings
router.post("/", async (req, res) => {
  const session = driver.session();
  const { name } = req.body;

  if (!name || name.trim().length < 2) return res.status(400).json({ error: "Název budovy
musí mít alespoň 2 znaky." });

  try {
    const check = await session.run(
      "MATCH (b:Building) WHERE b.name = $name RETURN b",
      { name: name.trim() }
    );

    if (check.records.length > 0) {
      return res.status(409).json({ error: "Budova se stejným názvem již existuje." });
    }

    const idResult = await session.run(`
      MATCH (b:Building)
      RETURN coalesce(max(b.id), 0) + 1 AS nextId
    `);
    const nextId = toNum(idResult.records[0].get("nextId"));

    const result = await session.run(`
      CREATE (b:Building {id: $id, name: $name})
      RETURN b.id AS id, b.name AS name
    `, { id: nextId, name: name.trim() });

    const r = result.records[0];
    res.status(201).json({ id: toNum(r.get("id")), name: r.get("name") });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});
```

```
// PUT /buildings/:id
router.put("/:id", async (req, res) => {
  const session = driver.session();
  const id = Number(req.params.id);
  const { name } = req.body;

  if (isNaN(id)) return res.status(400).json({ error: "ID budovy musí být číslo." });
  if (!name || name.trim().length < 2) return res.status(400).json({ error: "Název budovy
musí mít alespoň 2 znaky." });

  try {
    const checkDuplicate = await session.run(
      "MATCH (b:Building) WHERE b.name = $name AND b.id <> $id RETURN b",
      { id, name: name.trim() }
    );

    if (checkDuplicate.records.length > 0) {
      return res.status(409).json({ error: "Budova se stejným názvem již existuje." });
    }

    const result = await session.run(`
      MATCH (b:Building {id: $id})
      SET b.name = $name
      RETURN b.id AS id, b.name AS name
    `, { id, name: name.trim() });

    if (result.records.length === 0) {
      return res.status(404).json({ error: "Budova s tímto ID nebyla nalezena." });
    }

    const r = result.records[0];
    res.json({ id: toNum(r.get("id")), name: r.get("name") });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// DELETE /buildings/:id
router.delete("/:id", async (req, res) => {
  const session = driver.session();
  const id = Number(req.params.id);

  if (isNaN(id)) return res.status(400).json({ error: "ID budovy musí být číslo." });

  try {
    const result = await session.run(`
      MATCH (b:Building {id: $id})
      OPTIONAL MATCH (b)-[:HAS_FLOOR]->(f:Floor)
      DETACH DELETE b, f
      RETURN count(b) as deletedCount
    `, { id });

    if (toNum(result.records[0].get("deletedCount")) === 0) {
      return res.status(404).json({ error: "Budova s tímto ID nebyla nalezena." });
    }

    res.json({ success: true, message: `Budova ID ${id} a její patra byla smazána.` });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
```

```
  }
});

// GET /buildings/:id/floors
router.get("/:id/floors", async (req, res) => {
  const session = driver.session();
  const buildingId = Number(req.params.id);

  if (isNaN(buildingId)) return res.status(400).json({ error: "ID budovy musí být číslo."
});

  try {
    const result = await session.run(`
      MATCH (b:Building {id: $buildingId})-[:HAS_FLOOR]->(f:Floor)
      RETURN b.id AS buildingId, b.name AS buildingName, f.id AS id, f.level AS level,
f.capacity AS capacity
      ORDER BY level
    `, { buildingId });

    if (result.records.length === 0) {
      const bCheck = await session.run("MATCH (b:Building {id: $buildingId}) RETURN b", {
buildingId });
      if (bCheck.records.length === 0) return res.status(404).json({ error: "Budova
nenalezena." });

      return res.json({ buildingId, floors: [] });
    }

    const first = result.records[0];
    res.json({
      building: { id: toNum(first.get("buildingId")), name: first.get("buildingName") },
      floors: result.records.map(r => ({
        id: toNum(r.get("id")),
        level: toNum(r.get("level")),
        capacity: toNum(r.get("capacity"))
      }))
    });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// POST /buildings/:id/floors
router.post("/:id/floors", async (req, res) => {
  const session = driver.session();
  const buildingId = Number(req.params.id);
  const { level, capacity } = req.body;

  if (isNaN(buildingId)) return res.status(400).json({ error: "ID budovy musí být číslo."
});
  if (isNaN(Number(level)) || Number(level) < 0) return res.status(400).json({ error: "Level
patra musí být nezáporné číslo." });
  if (isNaN(Number(capacity)) || Number(capacity) <= 0) return res.status(400).json({ error:
"Kapacita musí být kladné číslo." });

  try {
    const check = await session.run(`
      MATCH (b:Building {id: $buildingId})
      OPTIONAL MATCH (b)-[:HAS_FLOOR]->(f:Floor {level: $level})
      RETURN b, f
    `, { buildingId, level: Number(level) });
```

```
    if (check.records.length === 0) return res.status(404).json({ error: "Budova neexistuje."
});
    if (check.records[0].get("f") !== null) {
      return res.status(400).json({ error: `Patro na levelu ${level} již v této budově
existuje.` });
    }

    const floorId = buildingId * 100 + Number(level);

    const result = await session.run(`
      MATCH (b:Building {id:$buildingId})
      CREATE (f:Floor {id:$floorId, level:$level, capacity:$capacity})
      CREATE (b)-[:HAS_FLOOR]->(f)
      RETURN f.id AS id, f.level AS level, f.capacity AS capacity
    `, { buildingId, floorId, level: Number(level), capacity: Number(capacity) });

    const r = result.records[0];
    res.status(201).json({
      id: toNum(r.get("id")),
      level: toNum(r.get("level")),
      capacity: toNum(r.get("capacity"))
    });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

module.exports = router;
```

departmentRoutes.js

```
const express = require("express");
const { driver } = require("../db");
const { toNum } = require("../utils/neo4jUtils");

const router = express.Router();

// GET /departments
router.get("/", async (req, res) => {
  const session = driver.session();
  try {
    const result = await session.run(`
      MATCH (d:Department)
      OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
      OPTIONAL MATCH (d)-[:COLLABORATES_WITH]->(partner:Department)
      RETURN d.id AS id, d.name AS name, count(distinct p) AS people,
             partner.id AS partnerId, partner.name AS partnerName
      ORDER BY id
    `);
    res.json(result.records.map(r => ({
      id: toNum(r.get("id")),
      name: r.get("name"),
      people: toNum(r.get("people")),
      collaboratesWith: r.get("partnerId") ? {
        id: toNum(r.get("partnerId")),
        name: r.get("partnerName")
      } : null
    })));
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// POST /departments
router.post("/", async (req, res) => {
  const session = driver.session();
  const { name, collaboratesWithId } = req.body;

  if (!name || name.trim().length < 2) return res.status(400).json({ error: "Název je povinný
(min. 2 znaky)." });

  try {
    const check = await session.run("MATCH (d:Department {name: $name}) RETURN d", { name:
name.trim() });
    if (check.records.length > 0) return res.status(409).json({ error: "Oddělení s tímto
názvem již existuje." });

    const idResult = await session.run(`
      MATCH (d:Department)
      RETURN coalesce(max(d.id), 0) + 1 AS nextId
    `);
    const nextId = toNum(idResult.records[0].get("nextId"));

    const result = await session.run(`
      CREATE (d:Department {id: $id, name: $name})
      RETURN d.id AS id, d.name AS name
    `, { id: nextId, name: name.trim() });
```

```
    if (collaboratesWithId && !isNaN(Number(collaboratesWithId))) {
      await session.run(`
        MATCH (d1:Department {id: $id})
        MATCH (d2:Department {id: $partnerId})
        MERGE (d1)-[:COLLABORATES_WITH]->(d2)
        MERGE (d2)-[:COLLABORATES_WITH]->(d1)
      `, { id: nextId, partnerId: Number(collaboratesWithId) });
    }

    const created = result.records[0];
    res.status(201).json({
      id: toNum(created.get("id")),
      name: created.get("name"),
      message: "Oddělení úspěšně vytvořeno."
    });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// PUT /departments/:id/collaboration
router.put("/:id/collaboration", async (req, res) => {
  const session = driver.session();
  const deptId = Number(req.params.id);
  const { collaboratesWithId } = req.body;

  if (isNaN(deptId)) {
    return res.status(400).json({ error: "ID oddělení musí být číslo." });
  }

  try {
    if (collaboratesWithId === null || collaboratesWithId === undefined || collaboratesWithId
=== "") {
      await session.run(`
        MATCH (d:Department {id: $deptId})
        OPTIONAL MATCH (d)-[r1:COLLABORATES_WITH]->(other:Department)
        OPTIONAL MATCH (other)-[r2:COLLABORATES_WITH]->(d)
        DELETE r1, r2
      `, { deptId });
      return res.json({ success: true, message: "Spolupráce byla odebrána." });
    }

    const partnerId = Number(collaboratesWithId);
    if (isNaN(partnerId)) {
      return res.status(400).json({ error: "ID partnera musí být číslo." });
    }

    await session.run(`
      MATCH (d1:Department {id: $deptId})
      MATCH (d2:Department {id: $partnerId})
      OPTIONAL MATCH (d1)-[r1:COLLABORATES_WITH]->()
      OPTIONAL MATCH ()-[r2:COLLABORATES_WITH]->(d1)
      DELETE r1, r2
      WITH d1, d2
      MERGE (d1)-[:COLLABORATES_WITH]->(d2)
      MERGE (d2)-[:COLLABORATES_WITH]->(d1)
    `, { deptId, partnerId });

    res.json({ success: true, message: "Vztah spolupráce byl aktualizován." });
  } catch (err) {
    res.status(500).json({ error: err.message });
```

```
  } finally {
    await session.close();
  }
});

// PUT /departments/:id
router.put("/:id", async (req, res) => {
  const session = driver.session();
  const id = Number(req.params.id);
  const { name } = req.body;

  if (isNaN(id)) return res.status(400).json({ error: "ID oddělení musí být číslo." });
  if (!name || name.trim().length < 2) return res.status(400).json({ error: "Název oddělení
musí mít alespoň 2 znaky." });

  try {
    const checkDuplicate = await session.run(
      "MATCH (d:Department) WHERE d.name = $name AND d.id <> $id RETURN d",
      { id, name: name.trim() }
    );

    if (checkDuplicate.records.length > 0) {
      return res.status(409).json({ error: "Oddělení se stejným názvem již existuje." });
    }

    const result = await session.run(`
      MATCH (d:Department {id: $id})
      SET d.name = $name
      RETURN d.id AS id, d.name AS name
    `, { id, name: name.trim() });

    if (result.records.length === 0) {
      return res.status(404).json({ error: "Oddělení s tímto ID nebylo nalezeno." });
    }

    const r = result.records[0];
    res.json({ id: toNum(r.get("id")), name: r.get("name") });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// DELETE /departments/:id
router.delete("/:id", async (req, res) => {
  const session = driver.session();
  const id = Number(req.params.id);

  if (isNaN(id)) return res.status(400).json({ error: "ID oddělení musí být číslo." });

  try {
    await session.run(`
      MATCH (pl:Placement)-[:OF_DEPARTMENT]->(d:Department {id: $id})
      DETACH DELETE pl
    `, { id });

    const result = await session.run(`
      MATCH (d:Department {id: $id})
      DETACH DELETE d
      RETURN count(d) AS deletedCount
    `, { id });
```

```
    if (toNum(result.records[0].get("deletedCount")) === 0) {
      return res.status(404).json({ error: "Oddělení nebylo nalezeno." });
    }

    res.json({ success: true, message: `Oddělení s ID ${id} bylo smazáno.` });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

module.exports = router;
```

floorRoutes.js

```
const express = require("express");
const { driver } = require("../db");
const { toNum } = require("../utils/neo4jUtils");

const router = express.Router();

// PUT /floors/:id - Update floor capacity
router.put("/:id", async (req, res) => {
  const session = driver.session();
  const floorId = Number(req.params.id);
  const capacity = Number(req.body.capacity);

  if ([floorId, capacity].some(Number.isNaN)) {
    return res.status(400).json({ error: "floorId and capacity must be numbers." });
  }

  try {
    const result = await session.run(
      `
      MATCH (f:Floor {id:$floorId})
      SET f.capacity = $capacity
      RETURN f.id AS id, f.level AS level, f.capacity AS capacity
      `,
      { floorId, capacity }
    );

    if (result.records.length === 0) {
      return res.status(404).json({ error: "Floor not found." });
    }

    const r = result.records[0];
    res.json({
      id: toNum(r.get("id")),
      level: toNum(r.get("level")),
      capacity: toNum(r.get("capacity")),
    });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// DELETE /floors/:id - Delete a floor and its relationships
router.delete("/:id", async (req, res) => {
  const session = driver.session();
  const floorId = Number(req.params.id);

  try {
    const result = await session.run(
      `
      MATCH (f:Floor {id:$floorId})
      WITH f
      OPTIONAL MATCH (f)<-[r]-()
      DELETE r
      WITH f
      DETACH DELETE f
      RETURN $floorId AS deletedId
      `,
```

```
      { floorId }
    );

    if (result.records.length === 0) {
      return res.status(404).json({ error: "Floor not found." });
    }

    res.json({ deletedId: floorId });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

module.exports = router;
```

mapRoutes.js

```
const express = require("express");
const { driver } = require("../db");
const { toNum } = require("../utils/neo4jUtils");

const router = express.Router();

// GET /map - Celkový stav
router.get("/", async (req, res) => {
  const session = driver.session();
  try {
    const aRes = await session.run(`MATCH (a:Assignment) RETURN a.id AS id ORDER BY
a.createdAt DESC LIMIT 1`);
    const assignmentId = aRes.records.length ? toNum(aRes.records[0].get("id")) : null;

    const bfRes = await session.run(`
      MATCH (b:Building)
      OPTIONAL MATCH (b)-[:HAS_FLOOR]->(f:Floor)
      RETURN b.id AS buildingId, b.name AS buildingName, f.id AS floorId, f.level AS level,
f.capacity AS capacity
      ORDER BY b.id, f.level
    `);

    const buildingById = new Map();
    for (const r of bfRes.records) {
      const bId = toNum(r.get("buildingId"));
      if (!buildingById.has(bId)) buildingById.set(bId, { id: bId, name:
r.get("buildingName"), floors: [] });
      const fId = r.get("floorId");
      if (fId !== null) {
        buildingById.get(bId).floors.push({ id: toNum(fId), level: toNum(r.get("level")),
capacity: toNum(r.get("capacity")), occupied: 0, departments: [] });
      }
    }
    const buildings = Array.from(buildingById.values());
    const floorById = new Map();
    buildings.forEach(b => b.floors.forEach(f => floorById.set(f.id, f)));

    if (assignmentId !== null && floorById.size > 0) {
      const plRes = await session.run(`
        MATCH (a:Assignment {id: $assignmentId})-[:HAS_PLACEMENT]->(pl:Placement)-
[:OF_DEPARTMENT]->(d:Department), (pl)-[:ON_FLOOR]->(f:Floor)
        OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
        RETURN f.id AS floorId, d.id AS deptId, d.name AS deptName, count(p) AS people,
coalesce(pl.locked, false) AS locked
      `, { assignmentId });
      for (const r of plRes.records) {
        const floor = floorById.get(toNum(r.get("floorId")));
        if (floor) {
          const size = toNum(r.get("people"));
          floor.departments.push({ id: toNum(r.get("deptId")), name: r.get("deptName"),
size, locked: !!r.get("locked") });
          floor.occupied += size;
        }
      }
    }

    const unassignedQuery = assignmentId === null
      ? `MATCH (d:Department)
         OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
```

```
        OPTIONAL MATCH (d)-[:COLLABORATES_WITH]->(partner:Department)
        RETURN d.id AS id, d.name AS name, count(p) AS size,
               partner.id AS collaboratesWithId, partner.name AS collaboratesWithName
        ORDER BY id`
      : `MATCH (d:Department)
        WHERE NOT EXISTS {
          MATCH (a:Assignment {id: $assignmentId})-[:HAS_PLACEMENT]->(pl:Placement)-
[:OF_DEPARTMENT]->(d)
        }
        OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
        OPTIONAL MATCH (d)-[:COLLABORATES_WITH]->(partner:Department)
        RETURN d.id AS id, d.name AS name, count(p) AS size,
               partner.id AS collaboratesWithId, partner.name AS collaboratesWithName
        ORDER BY id`;
    const unassignedRes = await session.run(unassignedQuery, { assignmentId });
    const unassignedDepartments = unassignedRes.records.map(r => ({
      id: toNum(r.get("id")),
      name: r.get("name"),
      size: toNum(r.get("size")),
      collaboratesWith:         r.get("collaboratesWithId")         ?         {         id:
toNum(r.get("collaboratesWithId")), name: r.get("collaboratesWithName") } : null
    }));

    res.json({ assignmentId, buildings, unassignedDepartments });
  } catch (err) { res.status(500).json({ error: err.message }); }
  finally { await session.close(); }
});

// GET /map/:buildingId - Detail jedné budovy + Volné týmy
router.get("/:buildingId", async (req, res) => {
  const session = driver.session();
  const buildingId = Number(req.params.buildingId);
  try {
    const aRes = await session.run(`MATCH (a:Assignment) RETURN a.id AS id ORDER BY
a.createdAt DESC LIMIT 1`);
    const assignmentId = aRes.records.length ? toNum(aRes.records[0].get("id")) : null;

    const bRes = await session.run(`MATCH (b:Building {id: $buildingId}) RETURN b.id AS id,
b.name AS name`, { buildingId });
    if (bRes.records.length === 0) return res.status(404).json({ error: "Budova nenalezena."
});

    const fRes = await session.run(`MATCH (b:Building {id:$buildingId})-[:HAS_FLOOR]-
>(f:Floor) RETURN f.id AS id, f.level AS level, f.capacity AS capacity ORDER BY level`, {
buildingId });
    const floors = fRes.records.map(r => ({ id: toNum(r.get("id")), level:
toNum(r.get("level")), capacity: toNum(r.get("capacity")), occupied: 0, departments: [] }));
    const floorById = new Map(floors.map(f => [f.id, f]));

    if (assignmentId !== null && floors.length > 0) {
      const plRes = await session.run(`
        MATCH (a:Assignment {id: $assignmentId})-[:HAS_PLACEMENT]->(pl:Placement)-
[:OF_DEPARTMENT]->(d:Department), (pl)-[:ON_FLOOR]->(f:Floor)<-[:HAS_FLOOR]-(b:Building
{id:$buildingId})
        OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
        RETURN f.id AS floorId, d.id AS deptId, d.name AS deptName, count(p) AS people,
coalesce(pl.locked, false) AS locked
      `, { assignmentId, buildingId });
      for (const r of plRes.records) {
        const floor = floorById.get(toNum(r.get("floorId")));
        if (floor) {
          const size = toNum(r.get("people"));
```

```
          floor.departments.push({  id:  toNum(r.get("deptId")),  name:  r.get("deptName"),
size, locked: !!r.get("locked") });
          floor.occupied += size;
        }
      }
    }

    const unassignedQuery = assignmentId === null
      ? `MATCH (d:Department)
         OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
         OPTIONAL MATCH (d)-[:COLLABORATES_WITH]->(partner:Department)
         RETURN d.id AS id, d.name AS name, count(p) AS size,
                partner.id AS collaboratesWithId, partner.name AS collaboratesWithName
         ORDER BY id`
      : `MATCH (d:Department)
         WHERE NOT EXISTS {
           MATCH   (a:Assignment   {id:   $assignmentId})-[:HAS_PLACEMENT]->(pl:Placement)-
[:OF_DEPARTMENT]->(d)
         }
         OPTIONAL MATCH (d)<-[:WORKS_IN]-(p:Person)
         OPTIONAL MATCH (d)-[:COLLABORATES_WITH]->(partner:Department)
         RETURN d.id AS id, d.name AS name, count(p) AS size,
                partner.id AS collaboratesWithId, partner.name AS collaboratesWithName
         ORDER BY id`;
    const unassignedRes = await session.run(unassignedQuery, { assignmentId });
    const unassignedDepartments = unassignedRes.records.map(r => ({
      id: toNum(r.get("id")),
      name: r.get("name"),
      size: toNum(r.get("size")),
      collaboratesWith:         r.get("collaboratesWithId")         ?         {         id:
toNum(r.get("collaboratesWithId")), name: r.get("collaboratesWithName") } : null
    }));

    res.json({  assignmentId,  building:  {  id:  toNum(bRes.records[0].get("id")),  name:
bRes.records[0].get("name") }, floors, unassignedDepartments });
  } catch (err) { res.status(500).json({ error: err.message }); }
  finally { await session.close(); }
});

module.exports = router;
```

personRoutes.js

```
const express = require("express");
const { driver } = require("../db");
const { toNum } = require("../utils/neo4jUtils");

const router = express.Router();

// GET /persons
router.get("/", async (req, res) => {
  const session = driver.session();
  try {
    const aRes = await session.run(`MATCH (a:Assignment) RETURN a.id AS id ORDER BY
a.createdAt DESC LIMIT 1`);
    const latestAssignmentId = aRes.records.length > 0 ? toNum(aRes.records[0].get("id")) :
null;

    let result;
    if (latestAssignmentId !== null) {
      result = await session.run(`
        MATCH (p:Person)
        OPTIONAL MATCH (p)-[:WORKS_IN]->(d:Department)
        OPTIONAL MATCH (a:Assignment {id: $assignmentId})-[:HAS_PLACEMENT]->(pl:Placement)-
[:OF_DEPARTMENT]->(d)
        OPTIONAL MATCH (pl)-[:ON_FLOOR]->(f:Floor)<-[:HAS_FLOOR]-(b:Building)
        RETURN p.id AS id, p.firstName AS firstName, p.surname AS surname,
               d.id AS departmentId, d.name AS departmentName,
               b.id AS buildingId, b.name AS buildingName,
               f.id AS floorId, f.level AS floorLevel
        ORDER BY id
      `, { assignmentId: latestAssignmentId });
    } else {
      result = await session.run(`
        MATCH (p:Person)
        OPTIONAL MATCH (p)-[:WORKS_IN]->(d:Department)
        RETURN p.id AS id, p.firstName AS firstName, p.surname AS surname,
               d.id AS departmentId, d.name AS departmentName,
               null AS buildingId, null AS buildingName,
               null AS floorId, null AS floorLevel
        ORDER BY id
      `);
    }

    res.json(result.records.map(r => ({
      id: toNum(r.get("id")),
      firstName: r.get("firstName"),
      surname: r.get("surname"),
      department: r.get("departmentId") ? {
        id: toNum(r.get("departmentId")),
        name: r.get("departmentName")
      } : null,
      building: r.get("buildingId") ? {
        id: toNum(r.get("buildingId")),
        name: r.get("buildingName")
      } : null,
      floor: r.get("floorId") ? {
        id: toNum(r.get("floorId")),
        level: toNum(r.get("floorLevel"))
      } : null
    })));
  } catch (err) {
```

```
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// POST /persons
router.post("/", async (req, res) => {
  const session = driver.session();
  const { firstName, surname, departmentId } = req.body;

  if (!firstName || firstName.trim().length < 2) return res.status(400).json({ error: "Jméno
je povinné." });
  if (!surname || surname.trim().length < 2) return res.status(400).json({ error: "Příjmení
je povinné." });

  try {
    const idResult = await session.run(`
      MATCH (p:Person)
      RETURN coalesce(max(p.id), 0) + 1 AS nextId
    `);
    const nextId = toNum(idResult.records[0].get("nextId"));

    if (departmentId && !isNaN(Number(departmentId))) {
      const checkDept = await session.run("MATCH (d:Department {id: $departmentId}) RETURN
d", { departmentId: Number(departmentId) });
      if (checkDept.records.length === 0) {
        return res.status(404).json({ error: "Cílové oddělení neexistuje." });
      }

      const result = await session.run(`
        MATCH (d:Department {id: $departmentId})
        CREATE (p:Person {id: $personId, firstName: $firstName, surname: $surname})
        CREATE (p)-[:WORKS_IN]->(d)
        RETURN p.id AS id, p.firstName AS firstName, p.surname AS surname,
               d.id AS departmentId, d.name AS departmentName
      `, {
        personId: nextId,
        firstName: firstName.trim(),
        surname: surname.trim(),
        departmentId: Number(departmentId)
      });

      const r = result.records[0];
      return res.status(201).json({
        id: toNum(r.get("id")),
        firstName: r.get("firstName"),
        surname: r.get("surname"),
        department: { id: toNum(r.get("departmentId")), name: r.get("departmentName") }
      });
    } else {
      const result = await session.run(`
        CREATE (p:Person {id: $personId, firstName: $firstName, surname: $surname})
        RETURN p.id AS id, p.firstName AS firstName, p.surname AS surname
      `, {
        personId: nextId,
        firstName: firstName.trim(),
        surname: surname.trim()
      });

      const r = result.records[0];
      return res.status(201).json({
        id: toNum(r.get("id")),
```

```
        firstName: r.get("firstName"),
        surname: r.get("surname"),
        department: null
      });
    }

  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// PUT /persons/:id/department
router.put("/:id/department", async (req, res) => {
  const session = driver.session();
  const personId = Number(req.params.id);
  const { departmentId } = req.body;

  if (isNaN(personId)) {
    return res.status(400).json({ error: "personId musí být číslo." });
  }

  try {
    if (departmentId === null) {
      const result = await session.run(`
        MATCH (p:Person {id: $personId})
        OPTIONAL MATCH (p)-[oldRel:WORKS_IN]->(:Department)
        DELETE oldRel
        RETURN p.id AS id, p.firstName AS firstName, p.surname AS surname
      `, { personId });

      if (result.records.length === 0) {
        return res.status(404).json({ error: "Osoba nenalezena." });
      }

      const r = result.records[0];
      return res.json({
        id: toNum(r.get("id")),
        firstName: r.get("firstName"),
        surname: r.get("surname"),
        department: null
      });
    } else {
      if (isNaN(Number(departmentId))) {
        return res.status(400).json({ error: "departmentId musí být číslo nebo null." });
      }

      const checkRel = await session.run(`
        MATCH (p:Person {id: $personId})-[:WORKS_IN]->(existing:Department)
        RETURN existing.id AS existingId, existing.name AS existingName
      `, { personId });

      if (checkRel.records.length > 0) {
        const existingId = toNum(checkRel.records[0].get("existingId"));
        const existingName = checkRel.records[0].get("existingName");
        if (existingId !== Number(departmentId)) {
          return res.status(409).json({
            error: `Zaměstnanec je již zařazen v oddělení "${existingName}". Nejdříve jej
z původního týmu odeberte.`
          });
        } else {
          return res.json({ message: "Zaměstnanec je již členem tohoto týmu." });
```

```
        }
      }

      const result = await session.run(`
        MATCH (p:Person {id: $personId})
        MATCH (d:Department {id: $departmentId})
        CREATE (p)-[:WORKS_IN]->(d)
        RETURN p.id AS id, p.firstName AS firstName, p.surname AS surname,
               d.id AS deptId, d.name AS deptName
      `, { personId, departmentId: Number(departmentId) });

      if (result.records.length === 0) {
        return res.status(404).json({ error: "Osoba nebo cílové oddělení nebylo nalezeno."
});
      }

      const r = result.records[0];
      return res.json({
        id: toNum(r.get("id")),
        firstName: r.get("firstName"),
        surname: r.get("surname"),
        department: { id: toNum(r.get("deptId")), name: r.get("deptName") }
      });
    }
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

// DELETE /persons/:id
router.delete("/:id", async (req, res) => {
  const session = driver.session();
  const personId = Number(req.params.id);

  if (isNaN(personId)) return res.status(400).json({ error: "ID osoby musí být číslo." });

  try {
    const result = await session.run(`
      MATCH (p:Person {id: $personId})
      DETACH DELETE p
      RETURN count(p) AS deletedCount
    `, { personId });

    if (toNum(result.records[0].get("deletedCount")) === 0) {
      return res.status(404).json({ error: "Osoba s tímto ID nebyla nalezena." });
    }

    res.json({ success: true, message: `Osoba s ID ${personId} byla smazána.` });
  } catch (err) {
    res.status(500).json({ error: err.message });
  } finally {
    await session.close();
  }
});

module.exports = router;
```

ilpSolver.js

```
const solver = require("javascript-lp-solver");

const COLLAB_WEIGHT = 1000;
const UNASSIGNED_PENALTY = 500;

/**
 * @param {object} params
 * @param {Array<{id:number, name:string, size:number}>} params.departments
 * @param {Array<{id:number, capacity:number, buildingName:string}>} params.floors
 * @param {Array<{deptId:number, floorId:number}>} params.lockedPlacements
 * @param {Array<{deptId:number, partnerId:number}>} params.collaborations
 *
 *  @returns  {{  assignments:  Array<{deptId:number,  floorId:number}>,  failed:  Array,
collaborationScore: number }}
 */
function solveAllocation({ departments, floors, lockedPlacements, collaborations }) {
  const lockedMap = new Map(lockedPlacements.map(p => [p.deptId, p.floorId]));

  const freeDepts = departments.filter(d => !lockedMap.has(d.id));

  const model = {
    optimize: "obj",
    opType: "max",
    constraints: {},
    variables: {},
    ints: {},
  };

  const xName = (dId, fId) => `x_${dId}_${fId}`;
  const yName = (dId, kId, fId) => `y_${Math.min(dId, kId)}_${Math.max(dId, kId)}_${fId}`;

  for (const d of freeDepts) {
    const cName = `assign_${d.id}`;
    model.constraints[cName] = { max: 1 };
    for (const f of floors) {
      const name = xName(d.id, f.id);
      if (!model.variables[name]) model.variables[name] = {};
      model.variables[name]["obj"] = 0;
      model.variables[name][cName] = 1;
      model.ints[name] = 1;
    }
  }

  for (const f of floors) {
    const lockedOccupied = lockedPlacements
      .filter(p => p.floorId === f.id)
      .reduce((sum, p) => {
        const d = departments.find(x => x.id === p.deptId);
        return sum + (d ? d.size : 0);
      }, 0);

    const remaining = f.capacity - lockedOccupied;
    const cName = `cap_${f.id}`;
    model.constraints[cName] = { max: remaining };

    for (const d of freeDepts) {
      const name = xName(d.id, f.id);
      if (!model.variables[name]) model.variables[name] = {};
      model.variables[name][cName] = d.size;
```

```
  }
}

for (const d of freeDepts) {
  const sName = `s_${d.id}`;
  const eqName = `eq_${d.id}`;

  model.constraints[eqName] = { min: 1, max: 1 };

  for (const f of floors) {
    const name = xName(d.id, f.id);
    if (!model.variables[name]) model.variables[name] = {};
    model.variables[name][eqName] = 1;
  }

  model.variables[sName] = {
    obj: -UNASSIGNED_PENALTY,
    [eqName]: 1,
  };
  model.ints[sName] = 1;
  model.constraints[`s_bound_${d.id}`] = { max: 1 };
  model.variables[sName][`s_bound_${d.id}`] = 1;
}

let fixedCollaborationScore = 0;
const semiLockedCollabs = [];

const collabPairs = [];
const seen = new Set();
for (const c of collaborations) {
  const aId = Math.min(c.deptId, c.partnerId);
  const bId = Math.max(c.deptId, c.partnerId);
  const key = `${aId}_${bId}`;
  if (!seen.has(key)) {
    seen.add(key);

    const aLockedFloor = lockedMap.get(aId);
    const bLockedFloor = lockedMap.get(bId);

    if (aLockedFloor !== undefined && bLockedFloor !== undefined) {
      // Oba locked
      if (aLockedFloor === bLockedFloor) fixedCollaborationScore++;
    } else if (aLockedFloor !== undefined || bLockedFloor !== undefined) {
      // Jeden locked, jeden volny
      const lockedFloor = aLockedFloor !== undefined ? aLockedFloor : bLockedFloor;
      const freeDeptId = aLockedFloor !== undefined ? bId : aId;
      semiLockedCollabs.push({ freeDeptId, floorId: lockedFloor });

      const name = xName(freeDeptId, lockedFloor);
      if (!model.variables[name]) model.variables[name] = {};
      model.variables[name]["obj"] = (model.variables[name]["obj"] || 0) + COLLAB_WEIGHT;
    } else {
      // Oba volni
      collabPairs.push({ a: aId, b: bId });
    }
  }
}

for (const pair of collabPairs) {
  for (const f of floors) {
    const yN = yName(pair.a, pair.b, f.id);
    const xA = xName(pair.a, f.id);
    const xB = xName(pair.b, f.id);
```

```
      model.variables[yN] = { obj: COLLAB_WEIGHT };
      model.ints[yN] = 1;

      // y <= x_a
      const c4 = `c4_${pair.a}_${pair.b}_${f.id}`;
      model.constraints[c4] = { max: 0 };
      model.variables[yN][c4] = 1;
      if (!model.variables[xA]) model.variables[xA] = {};
      model.variables[xA][c4] = -1;

      // y <= x_b
      const c5 = `c5_${pair.a}_${pair.b}_${f.id}`;
      model.constraints[c5] = { max: 0 };
      model.variables[yN][c5] = 1;
      if (!model.variables[xB]) model.variables[xB] = {};
      model.variables[xB][c5] = -1;
    }
  }

  for (const pair of collabPairs) {
    for (let j = 0; j < floors.length; j++) {
      for (let m = j + 1; m < floors.length; m++) {
        const fj = floors[j];
        const fm = floors[m];

        if (fj.buildingName !== fm.buildingName || Math.abs(fj.level - fm.level) > 1) {

          const c1 = `dist_${pair.a}_${pair.b}_${fj.id}_${fm.id}_1`;
          model.constraints[c1] = { max: 1 };
          if (!model.variables[xName(pair.a, fj.id)]) model.variables[xName(pair.a, fj.id)]
= {};
          if (!model.variables[xName(pair.b, fm.id)]) model.variables[xName(pair.b, fm.id)]
= {};
          model.variables[xName(pair.a, fj.id)][c1] = 1;
          model.variables[xName(pair.b, fm.id)][c1] = 1;

          const c2 = `dist_${pair.a}_${pair.b}_${fj.id}_${fm.id}_2`;
          model.constraints[c2] = { max: 1 };
          if (!model.variables[xName(pair.a, fm.id)]) model.variables[xName(pair.a, fm.id)]
= {};
          if (!model.variables[xName(pair.b, fj.id)]) model.variables[xName(pair.b, fj.id)]
= {};
          model.variables[xName(pair.a, fm.id)][c2] = 1;
          model.variables[xName(pair.b, fj.id)][c2] = 1;
        }
      }
    }
  }

  for (const semi of semiLockedCollabs) {
    const lockedFloor = floors.find(f => f.id === semi.floorId);
    if (!lockedFloor) continue;

    for (const f of floors) {
      if    (f.buildingName    !==    lockedFloor.buildingName    ||    Math.abs(f.level    -
lockedFloor.level) > 1) {
        const name = xName(semi.freeDeptId, f.id);
        if (!model.variables[name]) model.variables[name] = {};
        const cForbid = `forbid_${semi.freeDeptId}_${f.id}`;
        model.constraints[cForbid] = { max: 0 };
        model.variables[name][cForbid] = 1;
      }
```

```
    }
  }

  const result = solver.Solve(model);

  const assignments = [];

  for (const lp of lockedPlacements) {
    assignments.push({ deptId: lp.deptId, floorId: lp.floorId, locked: true });
  }

  const failed = [];
  for (const d of freeDepts) {
    let assigned = false;
    for (const f of floors) {
      const val = result[xName(d.id, f.id)];
      if (val && Math.round(val) === 1) {
        assignments.push({ deptId: d.id, floorId: f.id, locked: false });
        assigned = true;
        break;
      }
    }
    if (!assigned) {
      failed.push(d);
    }
  }

  let collaborationScore = fixedCollaborationScore;

  for (const semi of semiLockedCollabs) {
    const val = result[xName(semi.freeDeptId, semi.floorId)];
    if (val && Math.round(val) === 1) collaborationScore++;
  }

  for (const pair of collabPairs) {
    for (const f of floors) {
      const val = result[yName(pair.a, pair.b, f.id)];
      if (val && Math.round(val) === 1) { collaborationScore++; break; }
    }
  }

  const totalCollabPairs = fixedCollaborationScore + semiLockedCollabs.length +
collabPairs.length;
  return { assignments, failed, collaborationScore, totalCollabPairs };
}

module.exports = { solveAllocation };
```

neo4jUtils.js

```
const toNum = (v) => {
  if (v && typeof v.toNumber === "function") {
    return v.toNumber();
  }
  return v;
};
module.exports = { toNum };
```

app.js

```
const express = require("express");
const cors = require("cors");
require("dotenv").config();

// Import Routes
const mapRoutes = require("./routes/mapRoutes");
const buildingRoutes = require("./routes/buildingRoutes");
const departmentRoutes = require("./routes/departmentRoutes");
const personRoutes = require("./routes/personRoutes");
const floorRoutes = require("./routes/floorRoutes");
const assignmentRoutes = require("./routes/assignmentRoutes");
const { driver } = require("./db");

const app = express();
app.use(cors());
app.use(express.json());

// Health check
app.get("/health", async (req, res) => {
  const session = driver.session();
  try {
    const result = await session.run("RETURN 1 AS ok");
    res.json({ status: "ok", db: result.records[0].get("ok").toNumber() });
  } catch (err) { res.status(500).json({ error: err.message }); }
  finally { await session.close(); }
});

app.use("/map", mapRoutes);
app.use("/buildings", buildingRoutes);
app.use("/departments", departmentRoutes);
app.use("/persons", personRoutes);
app.use("/floors", floorRoutes);
app.use("/assignments", assignmentRoutes);

// 404 handler
app.use((req, res) => {
  res.status(404).json({ error: `Endpoint ${req.method} ${req.path} neexistuje.` });
});

// Global error handler
app.use((err, req, res, _next) => {
  console.error("[Global Error]", err);
  res.status(500).json({ error: err.message || "Interní chyba serveru." });
});

module.exports = app;
```

db.js

```
const neo4j = require("neo4j-driver");
require("dotenv").config();

const driver = neo4j.driver(
  process.env.NEO4J_URI,
  neo4j.auth.basic(
    process.env.NEO4J_USER,
    process.env.NEO4J_PASSWORD
  )
);

module.exports = { driver };
console.log("NEO4J_URI:", process.env.NEO4J_URI);
```

server.js

```
const express = require("express");
const cors = require("cors");
require("dotenv").config();
const { driver } = require("./db");

// Import Routes
const mapRoutes = require("./routes/mapRoutes");
const buildingRoutes = require("./routes/buildingRoutes");
const departmentRoutes = require("./routes/departmentRoutes");
const personRoutes = require("./routes/personRoutes");
const floorRoutes = require("./routes/floorRoutes");
const assignmentRoutes = require("./routes/assignmentRoutes"); // Nový import

const app = express();
app.use(cors());
app.use(express.json());

// Health check
app.get("/health", async (req, res) => {
  const session = driver.session();
  try {
    const result = await session.run("RETURN 1 AS ok");
    res.json({ status: "ok", db: result.records[0].get("ok").toNumber() });
  } catch (err) { res.status(500).json({ error: err.message }); }
  finally { await session.close(); }
});

// Register Domain Routes
app.use("/map", mapRoutes);
app.use("/buildings", buildingRoutes);
app.use("/departments", departmentRoutes);
app.use("/persons", personRoutes);
app.use("/floors", floorRoutes);
app.use("/assignments", assignmentRoutes); // Registrace nové cesty

// 404 handler – neznámé routes
app.use((req, res) => {
  res.status(404).json({ error: `Endpoint ${req.method} ${req.path} neexistuje.` });
});
```

```
// Global error handler
app.use((err, req, res, _next) => {
  console.error('[Global Error]', err);
  res.status(500).json({ error: err.message || 'Interní chyba serveru.' });
});

const port = Number(process.env.PORT || 3000);
app.listen(port, () => {
  console.log(`API running on http://localhost:${port}`);
});
```

assignments.test.js

```
const request = require("supertest");
const app = require("../src/app");
const { driver } = require("../src/db");

afterAll(async () => {
  await driver.close();
});

describe("Assignments API", () => {
  test("POST /assignments/clear – vyčistí mapu (vytvoří prázdný snapshot)", async () => {
    const res = await request(app).post("/assignments/clear");
    expect(res.status).toBe(200);
    expect(res.body.success).toBe(true);
    expect(res.body.assignmentId).toBeDefined();
  });

  test("POST /assignments/run – spustí automatickou alokaci", async () => {
    const res = await request(app).post("/assignments/run");
    expect([200, 400]).toContain(res.status);
    if (res.status === 200) {
      expect(res.body.success).toBe(true);
      expect(res.body.details).toBeDefined();
      expect(typeof res.body.details.totalAssigned).toBe("number");
      expect(Array.isArray(res.body.details.failedDepartments)).toBe(true);
    } else {
      expect(res.body.error).toBeDefined();
    }
  });

  test("POST /assignments/move – přesune tým na konkrétní patro", async () => {
    const depts = await request(app).get("/departments");
    const buildings = await request(app).get("/buildings");

    if (depts.body.length === 0 || buildings.body.length === 0) {
      console.warn("Test přeskočen: nejsou data pro přesun");
      return;
    }

    const firstDept = depts.body[0];
    const firstBuilding = buildings.body[0];
    const floorsRes = await request(app).get(`/buildings/${firstBuilding.id}/floors`);

    if (!floorsRes.body.floors || floorsRes.body.floors.length === 0) {
      console.warn("Test přeskočen: žádné patro v budově");
      return;
    }
```

```
    await request(app).post("/assignments/clear");

    const firstFloor = floorsRes.body.floors[0];
    const res = await request(app).post("/assignments/move").send({
      departmentId: firstDept.id,
      targetFloorId: firstFloor.id,
    });
    expect(res.status).toBe(200);
    expect(res.body.success).toBe(true);
  });

  test("POST /assignments/move – neexistující oddělení → 404", async () => {
    const res = await request(app).post("/assignments/move").send({
      departmentId: 999999,
      targetFloorId: 999999,
    });
    expect(res.status).toBe(404);
  });

  test("DELETE /assignments/:deptId/placement – odebere tým z mapy", async () => {
    const depts = await request(app).get("/departments");
    if (depts.body.length === 0) {
      console.warn("Test přeskočen: žádné oddělení");
      return;
    }
    await request(app).post("/assignments/run");

    const firstDept = depts.body[0];
    const res = await request(app).delete(`/assignments/${firstDept.id}/placement`);
    expect(res.status).toBe(200);
    expect(res.body.success).toBe(true);
  });
});
```

buildings.test.js

```
const request = require("supertest");
const { cleanupTestData, app, driver } = require("./testHelpers");

const uname = () => `Test_Budova_${Date.now()}_${Math.random().toString(36).slice(2, 6)}`;

beforeAll(async () => {
  await cleanupTestData();
});

afterAll(async () => {
  await cleanupTestData();
  await driver.close();
});

describe("Buildings API", () => {
  test("GET /buildings – vrátí pole budov", async () => {
    const res = await request(app).get("/buildings");
    expect(res.status).toBe(200);
    expect(Array.isArray(res.body)).toBe(true);
  });

  test("POST /buildings – vytvoří novou budovu", async () => {
    const name = uname();
    const res = await request(app).post("/buildings").send({ name });
    expect(res.status).toBe(201);
    expect(res.body.name).toBe(name);
    expect(res.body.id).toBeDefined();
  });

  test("POST /buildings – příliš krátký název → 400", async () => {
    const res = await request(app).post("/buildings").send({ name: "A" });
    expect(res.status).toBe(400);
    expect(res.body.error).toBeDefined();
  });

  test("POST /buildings – duplicitní název → 409", async () => {
    const name = uname();
    await request(app).post("/buildings").send({ name });
    const res = await request(app).post("/buildings").send({ name });
    expect(res.status).toBe(409);
  });

  test("PUT /buildings/:id – přejmenuje budovu", async () => {
    const name = uname();
    const create = await request(app).post("/buildings").send({ name });
    const id = create.body.id;
    const newName = uname() + "_renamed";
    const res = await request(app).put(`/buildings/${id}`).send({ name: newName });
    expect(res.status).toBe(200);
    expect(res.body.name).toBe(newName);
  });

  test("PUT /buildings/:id – neexistující ID → 404", async () => {
    const res = await request(app).put("/buildings/999999").send({ name: "Neco_validni" });
    expect(res.status).toBe(404);
  });

  test("DELETE /buildings/:id – smaže budovu", async () => {
    const name = uname();
```

```
    const create = await request(app).post("/buildings").send({ name });
    const id = create.body.id;
    const res = await request(app).delete(`/buildings/${id}`);
    expect(res.status).toBe(200);
    expect(res.body.success).toBe(true);
  });

  test("DELETE /buildings/:id — neexistující ID → 404", async () => {
    const res = await request(app).delete("/buildings/999999");
    expect(res.status).toBe(404);
  });

  test("POST /buildings/:id/floors — přidá patro", async () => {
    const name = uname();
    const create = await request(app).post("/buildings").send({ name });
    const bId = create.body.id;
    const  res  =  await  request(app).post(`/buildings/${bId}/floors`).send({  level:  1,
capacity: 20 });
    expect(res.status).toBe(201);
    expect(res.body.level).toBe(1);
    expect(res.body.capacity).toBe(20);
  });

  test("POST /buildings/:id/floors — duplicitní level → 400", async () => {
    const name = uname();
    const create = await request(app).post("/buildings").send({ name });
    const bId = create.body.id;
    await request(app).post(`/buildings/${bId}/floors`).send({ level: 1, capacity: 20 });
    const  res  =  await  request(app).post(`/buildings/${bId}/floors`).send({  level:  1,
capacity: 30 });
    expect(res.status).toBe(400);
  });
});
```

departments.test.js

```
const request = require("supertest");
const { cleanupTestData, app, driver } = require("./testHelpers");

const uname = () => `Test_Dept_${Date.now()}_${Math.random().toString(36).slice(2, 6)}`;

beforeAll(async () => {
  await cleanupTestData();
});

afterAll(async () => {
  await cleanupTestData();
  await driver.close();
});

describe("Departments API", () => {
  test("GET /departments – vrátí pole oddělení", async () => {
    const res = await request(app).get("/departments");
    expect(res.status).toBe(200);
    expect(Array.isArray(res.body)).toBe(true);
  });

  test("POST /departments – vytvoří nové oddělení", async () => {
    const name = uname();
    const res = await request(app).post("/departments").send({ name });
    expect(res.status).toBe(201);
    expect(res.body.id).toBeDefined();
    expect(res.body.name).toBe(name);
  });

  test("POST /departments – krátký název → 400", async () => {
    const res = await request(app).post("/departments").send({ name: "X" });
    expect(res.status).toBe(400);
  });

  test("POST /departments – duplicitní název → 409", async () => {
    const name = uname();
    await request(app).post("/departments").send({ name }); // první projde
    const res = await request(app).post("/departments").send({ name }); // druhý → 409
    expect(res.status).toBe(409);
  });

  test("PUT /departments/:id – přejmenuje oddělení", async () => {
    const name = uname();
    const create = await request(app).post("/departments").send({ name });
    const id = create.body.id;
    const newName = uname() + "_renamed";
    const res = await request(app).put(`/departments/${id}`).send({ name: newName });
    expect(res.status).toBe(200);
    expect(res.body.name).toBe(newName);
  });

  test("PUT /departments/:id/collaboration – nastaví spolupráci", async () => {
    const nameA = uname();
    const nameB = uname();
    const a = await request(app).post("/departments").send({ name: nameA });
    const b = await request(app).post("/departments").send({ name: nameB });
    const res = await request(app)
      .put(`/departments/${a.body.id}/collaboration`)
      .send({ collaboratesWithId: b.body.id });
```

```
    expect(res.status).toBe(200);
    expect(res.body.success).toBe(true);
  });

  test("PUT /departments/:id/collaboration s null — odebere spolupráci", async () => {
    const nameA = uname();
    const nameB = uname();
    const a = await request(app).post("/departments").send({ name: nameA });
    const b = await request(app).post("/departments").send({ name: nameB });
    await                request(app).put(`/departments/${a.body.id}/collaboration`).send({
collaboratesWithId: b.body.id });
    const res = await request(app)
      .put(`/departments/${a.body.id}/collaboration`)
      .send({ collaboratesWithId: null });
    expect(res.status).toBe(200);
    const check = await request(app).get("/departments");
    const dept = check.body.find(d => d.id === a.body.id);
    expect(dept?.collaboratesWith).toBeNull();
  });

  test("DELETE /departments/:id — smaže oddělení", async () => {
    const name = uname();
    const create = await request(app).post("/departments").send({ name });
    const res = await request(app).delete(`/departments/${create.body.id}`);
    expect(res.status).toBe(200);
    expect(res.body.success).toBe(true);
  });

  test("DELETE /departments/:id — neexistující → 404", async () => {
    const res = await request(app).delete("/departments/999999");
    expect(res.status).toBe(404);
  });
});
```

health.test.js

```
const request = require("supertest");
const { cleanupTestData, app, driver } = require("./testHelpers");

beforeAll(async () => {
  await cleanupTestData();
});

afterAll(async () => {
  await cleanupTestData();
  await driver.close();
});

describe("GET /health", () => {
  test("vrátí status ok", async () => {
    const res = await request(app).get("/health");
    expect(res.status).toBe(200);
    expect(res.body.status).toBe("ok");
  });
});

describe("404 handler", () => {
  test("neznámý endpoint vrátí 404", async () => {
    const res = await request(app).get("/neexistuje");
    expect(res.status).toBe(404);
    expect(res.body.error).toMatch(/neexistuje/);
  });
});
```

persons.test.js

```
const request = require("supertest");
const { cleanupTestData, app, driver } = require("./testHelpers");

beforeAll(async () => {
  await cleanupTestData();
});

afterAll(async () => {
  await cleanupTestData();
  await driver.close();
});

describe("Persons API", () => {
  test("GET /persons – vrátí pole zaměstnanců", async () => {
    const res = await request(app).get("/persons");
    expect(res.status).toBe(200);
    expect(Array.isArray(res.body)).toBe(true);
  });

  test("POST /persons – vytvoří zaměstnance bez oddělení", async () => {
    const res = await request(app).post("/persons").send({
      firstName: "Test",
      surname: `Uzivatel_${Date.now()}`,
    });
    expect(res.status).toBe(201);
    expect(res.body.id).toBeDefined();
    expect(res.body.department).toBeNull();
  });
```

```
  test("POST /persons – chybí jméno → 400", async () => {
    const res = await request(app).post("/persons").send({ surname: "Prijmeni" });
    expect(res.status).toBe(400);
  });

  test("POST /persons – příliš krátké příjmení → 400", async () => {
    const res = await request(app).post("/persons").send({ firstName: "Test", surname: "X"
});
    expect(res.status).toBe(400);
  });

  test("PUT /persons/:id/department – přiřadí zaměstnance do oddělení", async () => {
    const person = await request(app).post("/persons").send({
      firstName: "Test",
      surname: `Assign_${Date.now()}`,
    });
    const     dept     =     await     request(app).post("/departments").send({     name:
`Test_Dept_PA_${Date.now()}` });

    const res = await request(app)
      .put(`/persons/${person.body.id}/department`)
      .send({ departmentId: dept.body.id });
    expect(res.status).toBe(200);
    expect(res.body.department?.id).toBe(dept.body.id);
  });

  test("PUT /persons/:id/department – double-assign do jiného oddělení → 409", async () => {
    const person = await request(app).post("/persons").send({
      firstName: "Test",
      surname: `DoubleAssign_${Date.now()}`,
    });
    const     deptA     =     await     request(app).post("/departments").send({     name:
`Test_Dept_DA_${Date.now()}` });
    const     deptB     =     await     request(app).post("/departments").send({     name:
`Test_Dept_DB_${Date.now()}` });

    await   request(app).put(`/persons/${person.body.id}/department`).send({   departmentId:
deptA.body.id });
    const   res   =   await   request(app).put(`/persons/${person.body.id}/department`).send({
departmentId: deptB.body.id });
    expect(res.status).toBe(409);
    expect(res.body.error).toMatch(/již zařazen/);
  });

  test("PUT /persons/:id/department s null – odebere z oddělení", async () => {
    const person = await request(app).post("/persons").send({
      firstName: "Test",
      surname: `Unassign_${Date.now()}`,
    });
    const     dept     =     await     request(app).post("/departments").send({     name:
`Test_Dept_UN_${Date.now()}` });
    await   request(app).put(`/persons/${person.body.id}/department`).send({   departmentId:
dept.body.id });

    const   res   =   await   request(app).put(`/persons/${person.body.id}/department`).send({
departmentId: null });
    expect(res.status).toBe(200);
    expect(res.body.department).toBeNull();
  });

  test("DELETE /persons/:id – smaže zaměstnance", async () => {
    const person = await request(app).post("/persons").send({
```

```
      firstName: "Test",
      surname: `Delete_${Date.now()}`,
    });
    const res = await request(app).delete(`/persons/${person.body.id}`);
    expect(res.status).toBe(200);
    expect(res.body.success).toBe(true);
  });

  test("DELETE /persons/:id – neexistující → 404", async () => {
    const res = await request(app).delete("/persons/999999");
    expect(res.status).toBe(404);
  });
});
```

testHelpers.js

```
const request = require("supertest");
const app = require("../src/app");
const { driver } = require("../src/db");

async function cleanupTestData() {
  const session = driver.session();
  try {
    // smazani testovacich oddeleni
    await session.run(`
      MATCH (d:Department)
      WHERE d.name STARTS WITH 'Test_'
      DETACH DELETE d
    `);

    // smazani testovaci osoby
    await session.run(`
      MATCH (p:Person)
      WHERE p.firstName = 'Test'
      DETACH DELETE p
    `);

    // Smazani testovaci budovy
    await session.run(`
      MATCH (b:Building)
      WHERE b.name STARTS WITH 'Test_'
      OPTIONAL MATCH (b)-[:HAS_FLOOR]->(f:Floor)
      DETACH DELETE b, f
    `);
  } finally {
    await session.close();
  }
}

module.exports = { cleanupTestData, app, driver };
```

Databáze

```
CREATE CONSTRAINT building_id IF NOT EXISTS
FOR (b:Building) REQUIRE b.id IS UNIQUE;

CREATE CONSTRAINT floor_id IF NOT EXISTS
FOR (f:Floor) REQUIRE f.id IS UNIQUE;

CREATE CONSTRAINT department_id IF NOT EXISTS
FOR (d:Department) REQUIRE d.id IS UNIQUE;

CREATE CONSTRAINT person_id IF NOT EXISTS
FOR (p:Person) REQUIRE p.id IS UNIQUE;

CREATE CONSTRAINT assignment_id IF NOT EXISTS
FOR (a:Assignment) REQUIRE a.id IS UNIQUE;

CREATE CONSTRAINT placement_id IF NOT EXISTS
FOR (pl:Placement) REQUIRE pl.id IS UNIQUE;
```