**UNIVERZITA OBRANY V BRNĚ**

**FAKULTA VOJENSKÝCH TECHNOLOGIÍ**

**Studijní program: Technologie pro obranu a bezpečnost**

Studijní obor: Komunikační a Informační Technologie

|  |  |  |
| --- | --- | --- |
| Ev. číslo: |  | Počet listů: 88 |

**DIPLOMOVÁ PRÁCE**

Název: **INFORMAČNÍ SYSTÉM S AUDITNÍ STOPOU PRO PODPORU PROCESNÍHO ŘÍZENÍ ADMINISTRATIVY**

Zpracoval: Bc. Pham Si Anh

Vedoucí práce: Prof. Dr. Ing. Alexandr Štefek

BRNO 2020

Copy of thesis assignmentsAcknowledgement

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci jsem zpracoval samostatně pod vedením vedoucího bakalářského práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou uvedeny v seznamu literatury.

Brno 06.04.2022

|  |  |  |
| --- | --- | --- |
| ……………………….  Hodnost |  | ………………………………………  Jméno, přímení |

Abstrakt

Cílem této práce je vytvořit informační systém (s přístupem přes web) s funkcionalitou podporující administrativní proces s prvky umožňujícími audit prováděných úkolů. Teoretická část popisuje stručný úvod do procesu administrace, API a tří typů webových API, technologického zásobníku používaného pro vývoj informačního systému. Praktická část pojednává o procesu administrace podrobněji, popisuje datový model s charakteristikami z diskuse, návrh API a jeho implementaci, technickou dokumentaci k API, minimalistické uživatelské rozhraní a spolupracující kontejnery s Docker Compose. Výstup této práce je poskytován v repositáři GitHub a v této části je také popsána jeho struktura.

Klíčová slova

Administrativní proces, audit, web, API, Docker, kontejner, Git, FastAPI, React

Abstract

The goal of this thesis is to create an information system (with access via web) with functionality supporting the administration process with elements enabling an audit of the performed tasks. Theoretical part describes a brief introduction about administration process, API and three types of web API, technology stack used for developing the information system. Practical part discusses about the administration process in more detail, describes the data model with characteristics from discussion, API design and its implementation, technical documentation for API, minimalist user interface, and cooperating containers with Docker Compose. The output of this thesis is provided in GitHub repository and its structure is also described in this part.

Keywords

Administration process, audit, web, API, Docker, container, Git, FastAPI, React.

Obsah

[Abbreviations 9](#_Toc99915612)

[Figures 11](#_Toc99915613)

[Tables 13](#_Toc99915614)

[1 Introduction 14](#_Toc99915615)

[1.1 Goal and assignments for this thesis 14](#_Toc99915616)

[1.2 What is administration process? 15](#_Toc99915617)

[1.3 What is audit and audit trail? 16](#_Toc99915618)

[1.4 Thesis’s structure 17](#_Toc99915619)

[2 Web API 19](#_Toc99915620)

[2.1 Application Programming Interface (API) 19](#_Toc99915621)

[2.2 Web-based application software and Web API 20](#_Toc99915622)

[2.3 Information system with access via web 20](#_Toc99915623)

[2.4 REST and RESTful API 21](#_Toc99915624)

[2.4.1 REST 21](#_Toc99915625)

[2.4.2 RESTful API 21](#_Toc99915626)

[2.5 RPC 23](#_Toc99915627)

[2.6 GraphQL 23](#_Toc99915628)

[2.7 Other type of web API 24](#_Toc99915629)

[2.8 Summary 24](#_Toc99915630)

[3 Technology stack 26](#_Toc99915631)

[3.1 Types of databases 26](#_Toc99915632)

[3.1.1 Relational database 26](#_Toc99915633)

[3.1.2 Non-relational database 27](#_Toc99915634)

[3.1.3 Choosing database 30](#_Toc99915635)

[3.2 FastAPI 31](#_Toc99915636)

[3.2.1 FastAPI 31](#_Toc99915637)

[3.2.2 SQLAlchemy 32](#_Toc99915638)

[3.2.3 Swagger specification and OpenAPI specification 32](#_Toc99915639)

[3.3 React 33](#_Toc99915640)

[3.4 Docker 33](#_Toc99915641)

[3.4.1 Docker and container 33](#_Toc99915642)

[3.4.2 Networking in Docker and Docker Compose 34](#_Toc99915643)

[3.5 Git and GitHub 36](#_Toc99915644)

[3.5.1 The inspiration from Git 37](#_Toc99915645)

[3.6 Summary 39](#_Toc99915646)

[4 Discussion and implementation 40](#_Toc99915647)

[4.1 Administration process 40](#_Toc99915648)

[4.1.1 Phases in administration process 41](#_Toc99915649)

[4.1.2 The designated position and director 44](#_Toc99915650)

[4.1.3 Structure of a form 45](#_Toc99915651)

[4.1.4 Instance and its transition 46](#_Toc99915652)

[4.1.5 Audit and audit trails 47](#_Toc99915653)

[4.2 Data model design 48](#_Toc99915654)

[4.2.1 Users, groups, roles, positions 50](#_Toc99915655)

[4.2.2 Administration process and form 52](#_Toc99915656)

[4.2.3 Instances of a form 56](#_Toc99915657)

[4.2.4 Audit and audit trail 58](#_Toc99915658)

[4.3 API design and implementation 62](#_Toc99915659)

[4.3.1 Creating API with FastAPI 62](#_Toc99915660)

[4.3.2 The pattern of resource, resource instance and related resource 63](#_Toc99915661)

[4.3.3 API architecture 64](#_Toc99915662)

[4.4 Technical documentation 68](#_Toc99915663)

[4.4.1 Low-level documentation 68](#_Toc99915664)

[4.4.2 High-level documentation 69](#_Toc99915665)

[4.5 Minimalist user interface 70](#_Toc99915666)

[4.6 Containers with Docker Compose 73](#_Toc99915667)

[4.7 The necessary output 76](#_Toc99915668)

[4.8 Summary 77](#_Toc99915669)

[Conclusion 79](#_Toc99915670)

[Reference 81](#_Toc99915671)

[1 Appendix 1](#_Toc99915672)

[1.1 High-level documentation for API 1](#_Toc99915673)

[1.1.1 Define a sequence of phases (administration process) 1](#_Toc99915674)

[1.1.2 Define responsible users for every phase in administration process 2](#_Toc99915675)

[1.1.3 Define a structure of form 3](#_Toc99915676)

[1.1.4 How to create an instance from defined form 4](#_Toc99915677)

[1.1.5 How to transit instance through administration process 5](#_Toc99915678)

[1.1.6 How to recover instance at any phase. 6](#_Toc99915679)

# Seznam použitých zkratek

|  |  |
| --- | --- |
| ISO | International Organization for Standardization |
| NIST | National Institute of Standards and Technology |
| CNSSI | Committee on National Security Systems Instruction |
| API | Application Programming Interface |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| HTTP | Hypertext Transfer Protocol |
| HTML | Hypertext Markup Language |
| CSS | Cascading Style Sheets |
| JS | JavaScript |
| IS | Information System |
| REST | REpresentational State Transfer |
| CRUD | Create, Read, Update, Delete operations |
| JSON | JavaScript Object Notation |
| DOM | Document Object Model |
| XML | eXtensible Markup Language |
| RPC | Remote Procedural Call |
| SQL | Structure Query Language |
| RDBMS | Relational Database Management System |
| ORM | Object Relational Mapper |
| CNM | Container Network Model |
| VCS | Version Control System |
| SHA | Secure Hash Algorithm |
| OS | Operating System |
| ERD | Entity Relational Diagram |
| OOP | Object Oriented Programming |
| DDL | Data Definition Language |
| DML | Data Manipulation Language |
| POF | Path Operation Function |

# Seznam obrázků

[Obrázek 1 - Příklad administračního procesu žádosti o schválení návrhu tématu závěrečné práce 16](#_Toc100064678)

[Obrázek 2 - Relační databáze pro uložení fází a přechodů 29](#_Toc100064679)

[Obrázek 3 - Interaktivní dokumentační systém automaticky vytvořený pomocí Swagger UI z openapi.json 31](#_Toc100064680)

[Obrázek 4 - Vizualiace fází a přechodů pomocí CytoscapeJS 33](#_Toc100064681)

[Obrázek 5 - Kontejnerizace vs. Virtualizace 34](#_Toc100064682)

[Obrázek 6 - Docker network záložen na CNM 35](#_Toc100064683)

[Obrázek 7 - Vizualizace blob, tree a commit 38](#_Toc100064684)

[Obrázek 8- Vizualizace historie commitu 39](#_Toc100064685)

[Obrázek 9 - Lineární administrativní process 42](#_Toc100064686)

[Obrázek 10 - Nelineární administrativní proces 44](#_Toc100064687)

[Obrázek 11 - Administrativní proces s řediteli 45](#_Toc100064688)

[Obrázek 12 - Struktura formuláře 47](#_Toc100064689)

[Obrázek 13 - ERD kardinalita 50](#_Toc100064690)

[Obrázek 14 - Datový model pro uživatelé, skupiny, role, pozice 52](#_Toc100064691)

[Obrázek 15 - Jednodušší (vlevo) a rozšířené (vpravo) řešení datového modelu administrativního procesu 54](#_Toc100064692)

[Obrázek 16 - Fáze před a po transformací 54](#_Toc100064693)

[Obrázek 17 - Přesun vztahu mezi formuláři a sekcemi na fáze a sekce 55](#_Toc100064694)

[Obrázek 18 - Datový model administrativního procesu 56](#_Toc100064695)

[Obrázek 19 - Datový model instance 57](#_Toc100064696)

[Obrázek 20 - Přidání Unique omezení pro tabulku contents explicitně v PostgreSQL 59](#_Toc100064697)

[Obrázek 21 - Vytvoření tabulku directors a handlers s Unique omezení na více sloupce v PostgreSQL 59](#_Toc100064698)

[Obrázek 22 - Vytvoření envelope 61](#_Toc100064699)

[Obrázek 23 - Příklad dvě commit 62](#_Toc100064700)

[Obrázek 24 - Datový model pro auditování 63](#_Toc100064701)

[Obrázek 25 - Interaktivní dokumentace vytvořen pomocí Swagger UI 64](#_Toc100064702)

[Obrázek 26 - Diagram třídy pro vytvoření API pomocí FastAPI 67](#_Toc100064703)

[Obrázek 27 - Příklad nízkoúrovňové dokumentace pro koncového bodu 70](#_Toc100064704)

[Obrázek 28 - Vytvoření fáze v uživatelském rozhraní 72](#_Toc100064705)

[Obrázek 29 - Příklad instance v uživatelském rozhraní 73](#_Toc100064706)

[Obrázek 30 - Přechod instance v uživatelském rozhraní 74](#_Toc100064707)

[Obrázek 31 - Historie commitů (auditních stop) instance 75](#_Toc100064708)

[Obrázek 32 - Bridge network vytvořenp pomocí Compose 76](#_Toc100064709)

[Obrázek 33 - Adresář build vytvořen pomocí React 76](#_Toc100064710)

[Obrázek 34 - Komunikace mezi klientem a kontejnery 78](#_Toc100064711)

# Seznam tabulek

[Tabulka 1 - Datové typy datového modelu 51](#_Toc100064712)

[Tabulka 2 - 6 typů koncového bodů 65](#_Toc100064713)

# Úvod

## Cíl a úkoly této práce

Naším cílem je vytvořit informační systém (s přístupem přes web) s funkcionalitou podporující elektronizaci administrativy s prvky umožňující provést audit nad provedenými úkony. Z tohoto cíle musíme specifikovat, jakou funkcionalitu musí náš systém podporovat pro elektronizovat administrativní proces a které provedené úkony jsou auditovány. Požadavky na systém na funkčnost podporující administrativní proces jsou následující:

* Musí být možné definovat sekvence fází (administrační proces) zkušeným uživatelem s správným oprávněním.
* Musí být možné definovat, kdo je/jsou odpovědný za každou fázi administrativního procesu.
* Musí být možné definovat strukturu formuláře zkušeným uživatelem s správným oprávněním.
* Žadatel musí být schopen vytvořit instanci z definovaného formuláře (dále jen instance).
* Instance musí být schopna projít administračním procesem (fází).

Provedeným úkonem v procesu administrace je přechod instance přes fáze. Vytvoření procesu, přiřazení odpovědné osoby/lidí pro jednotlivé fáze, vytvoření formuláře se nepovažuje za provedený úkon v administrativním procesu. Je považováno za vytvoření administrativního procesu. Požadavky na audit nad provedenými úkony jsou:

* V každé fázi administračního procesu musí být u instance zaznamenáno, co se v obsahu změnilo, kdo jej změnil, kdy se změní.
* Musí být možné obnovit instanci v jakékoli fázi.

Pokyny pro zpracování závěrečné práce:

* Informační systém koncipovat jako množinu spolupracujících kontejnerů (docker compose).
* Nastudovat tvorbu API (podle standardu OpenAPI / Swagger) pomocí knihovny FastAPI.
* Vytvořit datový model informačního systému.
* Provádět implementaci.
* Zpracovat technickou dokumentaci k API.
* Vytvořit minimalistické uživatelské rozhraní pro možnost demonstrace funkcionality.

Nezbytným výstupem je:

* Zdrojový kód (na platformě GITHUB) kontejneru poskytující službu autentizovaným uživatelům.
* Dokumentace k API (aplikační rozhraní), přes které je služba zpřístupněna.
* Zdrojový kód (na platformě GITHUB) kontejneru realizující uživatelské rozhraní.

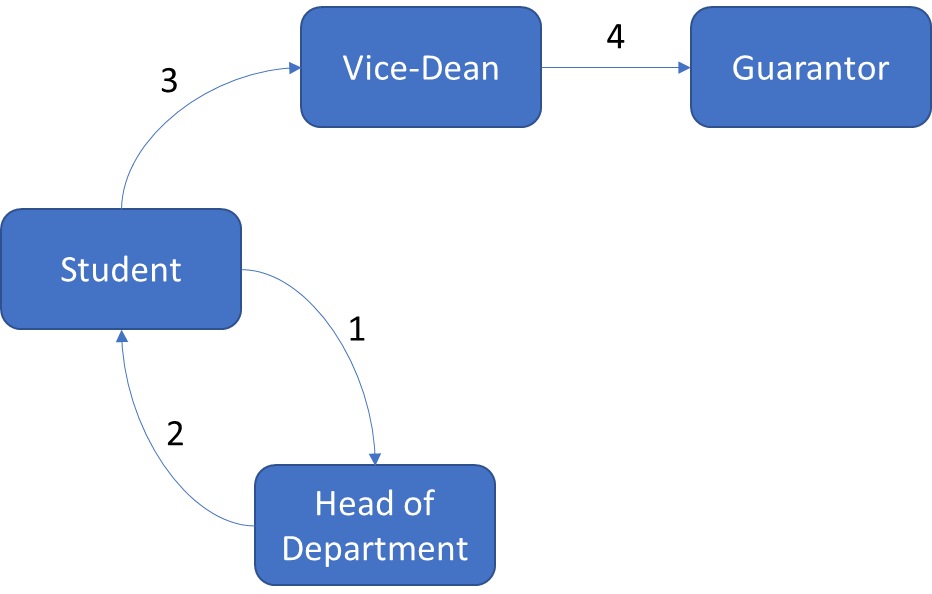
## Co je administrativní proces?

V této práci můžeme chápat administrativní proces jako soubor fází, kterými musí projít žádost žadatele. V každé fázi zodpovědná osoba/osoby vyřídí žádost žadatele a poté žádost odešlou do další fáze, kde ji opět vyřídí další osoba/osoby a pošlou ji do další fáze. Proces bude pokračovat, dokud nebude požadavek zcela vyřízen.

Dole je příklad administračního procesu, který vychází ze Studijního rozkazu děkana Fakulty vojenských technologií Univerzity obrany v Brně, jak student navrhuje vlastní téma závěrečné práce:

* Student Žádost o vlastní téma závěrečné práce předá vedoucímu odborné katedry k vyjádření (1). Vedoucí odborné katedry v souladu s Opatřením rektora č. 3/2018 Závěrečné práce posoudí naplnění požadavků na navržené téma závěrečné práce, cíl, popis řešené problematiky a zdůvodnění aktuálnosti tématu k předloženému návrhu. V návaznosti na navržené téma závěrečné práce doporučí vedoucího závěrečné práce.
* Žádost se stanoviskem vedoucího katedry (2) student doručí na Studijní skupinu děkanátu fakulty (3).
* Proděkanovi pro studijní a pedagogickou činnost (PdSPČ) si vyžádá stanovisko garanta studijního programu ke všem Žádostem o vlastní téma závěrečné práce s termínem vyjádření (4). V případě neschválení navrženého tématu nebo ostatních skutečností, které Žádost o vlastní téma závěrečné práce obsahuje je garant povinen uvést konkrétní důvody neschválení.

Proces si můžeme představit na Obrázek 1:



Obrázek 1 - Příklad administračního procesu žádosti o schválení návrhu tématu závěrečné práce

Zdroj: vlastní

Obecně bude proces definován předpísem a často i formulářem. Žadatel vytvoří instanci z tohoto formuláře a odešle ji osobě v další fázi. Procesu administrace se budeme podrobněji věnovat v kapitole 4.

## Co je audit a audit auditní stopa?

Audit je definován jako nezávislé zkoumání, inspekce, ověřování nebo přezkoumání. Předmět auditu může být v různých sektorech různý.

Termín audit se původně používá ve finančním sektoru. Antropologové našli záznamy o auditorské činnosti, které se datují do časných babylonských dob (kolem roku 3000 př.nl). Auditorská činnost byla také ve staré Číně, Řecku a Římě. Latinský význam slova „auditor“ byl „slyšící nebo posluchač“, protože v Římě auditoři slyšeli daňové poplatníky, jako jsou zemědělci, podávat veřejná prohlášení o výsledcích jejich podnikání a splatné daňové povinnosti. Praxe moderního auditu se datuje od počátku moderní korporace na úsvitu průmyslové revoluce v 18. století s růstem akciových společností a vlastnictvím a řízením se oddělily. [1] . Audit je definován prof. L.R. Dicksee jako „an examination of accounting records undertaken with a view to establish whether they correctly and completely reflect the transactions to which they relate.“

V sektoru informačních technologií definice auditu neomezuje ani nepředpokládá předmět, kterého se audit týká. Mezinárodní organizace pro standardizaci (ISO) definovala audit ve Guidelines for auditing management systems jako: “systematic, independent and documented process for obtaining objective evidence and evaluating it objectively to determine the extent to which the audit criteria are fulfilled.” [2] . NIST a CNSSI definují audit jako: “Independent review and examination of records and activities to assess the adequacy of system controls and ensure compliance with established policies and operational procedures.” [3] [4] .

NIST and CNSSI taky definují auditní stopa jako: “A chronological record that reconstructs and examines the sequence of activities surrounding or leading to a specific operation, procedure, or event in a security relevant transaction from inception to final result.” [3]  [4] .

V případě administračního procesu jsme určili, že předmětem auditu je požadavek žadatele a auditní stopa je chronologický záznam požadavku žadatele v každé fázi administračního procesu, který dokážeme rekonstruovat a prověřit sekvenci fází, kterými požadavek prošel v předdefinovaném administrativním procesu.

## Struktura diplomové práce

V kapitolu 2 definujeme, co je API, webové API a informační systém s přístupem přes web. Představíme také 3 typy webového API a probereme výhody a nevýhody těchto 3 typů. Myšlenka o zdrojích REST bude v této práci použita k implementaci API spolu s URL, HTTP a JSON..

V kapitolu 3 poskytujeme přehled o technologickém zásobníku, který se používá pro vývoj našeho informačního systému. Mezi tyto technologie patří PostgreSQL jako systém řízení relační databáze, FastAPI jako knihovna (framework) pro tvorbu API, React jako framework pro tvorbu uživatelského rozhraní, Docker jako virtualizační (kontejnerizační) technologie, Git a GitHub jako systém pro správu verzí a platforma pro hostování zdrojového kódu. Kromě těchto technologií najdeme v systému Git inspirativní myšlenku, kterou představíme v podkapitole The inspiration from Git.

Kapitol 4 je kde se zabýváme administrativním procesem. Každý požadavek popsán v oddílu 1.1 bude objasnit v oddílu 4.1. Provádění úkolů bude popsáno v oddílech 4.2 až 4.6. Nezbytný výstup je uveden v příloze a odkaz na platformu GitHub (podrobně v oddílu 4.7).

Conclusion shrnuje práci, její přínos, omezení a orientaci našeho systému pro další vývoj.

# Web API

## Aplikační programovací rozhraní (API)

Podle Blocha J. vznikla myšlenka aplikačního programového rozhraní z myšlenky knihovny podprogramů, když Maurice Wilkes a David Wheeler ve 40. letech 20. století vytvořili programy běžící v raném počítači s názvem EDSAC [6]. Klíčovou myšlenku podprogramu a způsob jeho dokumentace představil Wheeler v roce 1952 v článku *The Use of Sub-routines in Programmes*, ve kterém uvedl: “…it is usually advantageous to arrange that a programme is comprise of a set of subroutines some of which have been made specially for the particular programme while others are available from a 'library' of standard sub-routines” a “However, even after it has been coded and tested there still remains the considerable task of writing a description so that people not acquainted with the interior coding can nevertheless use it easily. This last task may be the most difficult.”

API je soubor definic a protokolů, jak spolu komunikují dva počítačové programy. API není implementace, ale specifikace. Zjednodušeně řečeno, počítačový program přijímá vstup, zpracovává jej a produkuje výstup, API pak popisuje, jak má vstup vypadat, jak má být vstup předán počítačovému programu a jak má vypadat výstup. Například procesor se svou instrukční sadou má API, operační systém se sadou systémových volání má API, programovací jazyk se sadou funkcí standardní knihovny nebo vestavěných funkcí má API.

První klíčovou myšlenkou API je skrytí vnitřní implementace počítačového programu tak, aby ji mohl jiný program nebo programátor snadno znovu použít nebo kombinovat. Druhou klíčovou myšlenkou API je možnost nezávislé reimplementace počítačového programu, aby neměla dopad na jiný program, který je na něm závislý, a to navzdory jakémukoli účelu reimplementace, ať už jde o jeho vylepšení nebo zhoršení.

Počítačový program, který implementuje rozhraní API a poskytuje je jinému počítačovému programu k použití, se nazývá poskytovatel API. Počítačový program, který API spotřebovává, se nazývá spotřebitel API. Poskytovatele a spotřebitele API lze také označit jako programátora, protože programátor je ten, kdo píše počítačový program pro poskytování nebo spotřebovávání jiného počítačového programu.

## Webový aplikační software a webové rozhraní API

Aplikační software je počítačový program, který řeší potřebu koncového uživatele pro určitý úkol, který lze vyřešit i bez použití tohoto programu. Například: psaní dokumentu lze řešit pomocí tužky a papíru, ale lze jej řešit pomocí textového procesoru; účetnictví a audit lze řešit pomocí tužky a papíru, ale lze jej řešit pomocí účetního a auditního softwaru; administrativní proces lze řešit pomocí tužky a papíru, ale lze jej řešit pomocí aplikačního softwaru, příležitostně webového softwaru, který se snažíme řešit v této práci.

Síťový aplikační software je aplikační software, který si vyměňuje svá data po síti. Specifičtějším typem síťového aplikačního softwaru je webový aplikační software (nebo jednoduše webová aplikace), který je vytvořen z technologií, na nichž je založen World Wide Web (nebo jednoduše web), např. URI, HTTP, HTML, ....

Webová aplikace implementující rozhraní API se nazývá webové rozhraní API.

## Informační systém s přístupem přes web

Informační systém (IS) je soubor vzájemně propojených prvků nebo komponent, které shromažďují (vstup), manipulují (zpracovávají), ukládají a šíří (výstup) data a informace a zajišťují korektivní reakci (mechanismus zpětné vazby) ke splnění cíle. Zpracování může probíhat ručně nebo s pomocí počítače. Počítačový informační systém je jednotný soubor hardwaru, softwaru, databází, telekomunikací, lidí a postupů, které jsou nakonfigurovány tak, aby shromažďovaly, manipulovaly, ukládaly a zpracovávaly data na informace [7]. Aplikační software je nejbližší částí softwarového prvku koncovému uživateli informačního systému. Mezi další části softwarového prvku patří operační systém, síťový software, ... se v této práci nesnažíme řešit, protože to není podstatná část problému.

Náš IS musí být přístupný přes web, jak je definováno v oddílu 1.1, proto aplikační software musí být webová aplikace. Webová aplikace musí také implementovat API a jeho technickou dokumentaci, jak je uvedeno v zadání této práce.

V dalších oddílech se budeme zabývat 3 nejpopulárnějšími typy požadavek–odpověď API či styly návrhu webového API [25].

## REST and RESTful API

### REST

Representational State Transfer (REST) je představen v kapitole 5 disertační práce Fieldinga R.T. z roku 2000 jako architektonický styl pro distribuované hypermediální [5]. REST is used for guiding the design and development of the architecture for the modern Web. REST se používá pro řízení návrhu a vývoje architektury pro moderní web. Fielding popsal REST jako soubor interakčních omezení mezi komponentami (např. klient-server, bezstavový, cache, jednotné rozhraní, vrstvený systém, kód na vyžádání), které nazývá stylem, s ohledem na inženýrské principy (např. princip oddělení obav, princip obecnosti), aby odvodil požadované architektonické vlastnosti moderní webové architektury (např. výkonnost, škálovatelnost, jednoduchost, modifikovatelnost, viditelnost, přenositelnost a spolehlivost).

Fielding R. T. také v 6. kapitole své disertační práce popisuje zkušenosti a poznatky získané při uplatňování REST při tvorbě internetových standardů pro protokol HTTP (Hypertext Transfer Protocol) a URI (Uniform Resource Identifiers). Tyto dvě specifikace definují obecné rozhraní, které se používá při interakci všech komponent na webu.

### RESTful API

V komunitě vývojářů webových aplikací se webové rozhraní pro programování aplikací (Web API nebo Web Service), které odpovídá architektonickému stylu REST, označuje jako REST API nebo RESTful API.

REST byl vyvinut jako model fungování moderního webu. V důsledku toho se odůvodnění REST používá pro průvodce standardy nebo technologiemi na webu, např. URI, HTTP, ...

Uniform Resource Identifier (URI) se používá k jednoznačné identifikaci zdroje, což je klíčová abstrakce informací v REST, v celém webu. Termín "Uniform Resource Locator" (URL) označuje podmnožinu URI, která kromě identifikace zdroje poskytuje prostředky pro jeho lokalizaci popisem jeho primárního přístupového mechanismu (např. jeho síťové " lokace") [8]. Termín URL je v kontextu této práce specifičtějším pojmem, proto budeme místo termínu URI používat termín URL. Na druhé straně zdrojem našeho IS může být jakákoli entita prezentovaná v našem systému, např. žadatel, formulář, instance, fáze, přechod, ....

Hypertext Transfer Protocol (HTTP) je jedním z mnoha komunikačních protokolů používaných na webu k výměně reprezentace zdroje. Má metody, např. GET, POST, PUT, PATCH, DELETE, ... definované v RFC 2616 [9], které umožňují interakci (CRUD) se zdrojem identifikovaným v URL. Tyto metody jsou dostatečně obecné pro interakci mezi konzumenty API a s téměř každým typem zdroje. Výjimkou je interakce zdroje s jiným(i) zdrojem(i), která vyžaduje specifický(é) parametr(y) než stavy samotného zdroje. Například přechod instance do další fáze vyžaduje informace o odpovědné osobě/osobách, což není stav samotné instance. Typickým problémem rozhraní REST API je také vyhledávání v celé kolekci zdrojů. Ačkoli interakci mezi zdroji můžeme považovat za jiný zdroj jako definici o zdroji v [8]: “… abstract concepts can be resources, such as the operators and operands of a mathematical equation …”, budeme směšovat pojem o zdroji a interakci a způsobíme zmatek a nejasnosti pro programátory, jako údržbář nebo spotřebitele.

Datový formát (media type) reprezentace zdroje v rozhraní REST API je často, ale ne vždy, JSON nebo XML[[1]](#footnote-1). V této práci používáme jako datový formát reprezentace zdroje JSON, protože má jednodušší syntaxi a menší režii než XML.

Použití stylu REST při návrhu rozhraní API jsou kontroverzním tématem [22] [23], kdy rozhraní REST API musí být hypertextově orientované. Fielding vysvětluje, že "REST is software design on the scale of decades: every detail is intended to promote software longevity and independent evolution. Many of the constraints are directly opposed to short-term efficiency.” [22]

Omezení v čase a lidských zdrojích vyžaduje větší úsilí při aplikaci REST v návrhu našeho API, aby bylo RESTful. Abychom urychlili proces studia tvorby API a implementace API, nebude se návrh API řídit omezením REST při použití hypertextu. Bude však uplatněna myšlenka o zdrojích. K vytvoření webového rozhraní API budou v této práci použity také adresy URL, HTTP a JSON.

Výhody:

* Snadná implementace, údržba a rozšiřování.
* Vhodné pro CRUD zdrojů a zobrazení vztahů mezi zdroji.

Nevýhody:

* Nevhodné pro konkrétní akci zdroje.
* Velký obsah v odpovědi.
* Pokud chceme získat reprezentaci více zdrojů, je nutné provést více oběhu HTTP.

## RPC

Vzdálené procedurální volání (RPC) V distribuovaných systémech je to situace, kdy klientský program volá funkci, jejíž implementace je na vzdáleném serveru. Ve webovém rozhraní API se rozhraní API ve stylu RPC zaměřuje na akce. Každý koncový bod představuje akci, kterou může klient provést na serveru. Akce je funkce, která je volána na serveru. Podobně jako běžná funkce, která přijímá parametry a vrací hodnotu, která je odeslána jako odpověď klientovi. U rozhraní API RPC, které používá jako transportní protokol HTTP, je metoda nebo funkce umístěna v adrese URL a argumenty jsou umístěny buď v řetězci dotazu, nebo v těle dotazu.

Styl RPC je vhodný pro rozhraní API, která vystavují různé akce zdrojů, které mohou mít více nuancí a komplikací než lze zapouzdřit pomocí CRUD. RPC částečně vyřešil problém REST, ale přináší nový problém. Každý typ zdroje bude mít nějakou společnou akci a nějakou vlastní specifickou akci. Nejprve opakovaně implementujeme společné akce na každém typu prostředku. Za druhé, není ideální standardizovat, spravovat, udržovat nebo spotřebovávat takto různorodý seznam akcí, protože tento seznam se bude časem rozšiřovat, až budeme chtít přidat další akci nebo další typ zdroje s vlastní specifickou akcí.

Výhody:

* Vhodné pro specifické akce na zdroji.

Nevýhody:

* Pouze pro konkrétní řešení, chybí standardizace.
* Může vést k explozi funkcí, pokud se přidá více akcí.

## GraphQL

GraphQL je dotazovací jazyk pro rozhraní API, který byl interně vyvinut společností Facebook v roce 2012 a veřejně zveřejněn v roce [10] a byl přijat poskytovateli rozhraní API, jako jsou GitHub, Yelp a Pinterest.

GraphQL umožňuje spotřebiteli definovat strukturu požadovaných dat a server vrátí data přesně s touto strukturou, čímž se sníží množství nadbytečných dat v odpovědi. GraphQL má jediný koncový bod pro interakci a dva typy operací, query pro načítání dat pouze pro čtení a mutation pro zápis s následným načtením.

GraphQL snižuje redundantní data v užitečném obsahu odpovědi a počet oběhů HTTP ve srovnání s rozhraním REST API. Je vhodný pro zdroje s velkým množstvím datových polí v jejich stavu a pro zdroje ve vysokém hierarchickém vztahu s jinými zdroji.

Nevýhodou jazyka GraphQL je složitost implementace, dodatečné zpracování při analýze dotazů a ověřování parametrů. Proto není jazyk GraphQL vhodný pro jednoduché rozhraní API. Nicméně jazyk GraphQL lze použít v kombinaci s rozhraním REST API a lze o něm uvažovat, až se systém stane větším. Pro jednoduchost našeho systému při implementaci v této práci jazyk GraphQL nepoužíváme, ale uvažujeme o něm jako o potenciálním řešení, až se náš systém vyvine a bude složitější.

Výhody:

* Méně nadbytečných dat v užitečném zatížení odpovědi
* Méně objíždění protokolu HTTP

Nevýhody:

* Další složitost
* Další zpracování
* Nevhodné pro jednoduché API

## Jiné typy webových rozhraní API

Mezi další typy webových API patří SOAP API, které se již nepoužívá; webové API řízené událostmi, např. Webhook pro komunikaci mezi serverem a serverem, WebSocket pro obousměrnou komunikaci mezi klientem a serverem v reálném čase, HTTP Streaming pro zasílání nekonečného množství dat v nekonečném počtu odpovědí; jsou pro požadavky našeho IS považovány za zbytečné, proto se jimi v této práci nebudeme dále zabývat. Nicméně je možné tyto typy webového API kombinovat s jinými typy webového API, pokud budou na náš IS kladeny další požadavky.

## Shrnutí

V této kapitole jsme se zabývali tím, co je to API, webové API, IS s přístupem přes web a třemi typy webových API, které se nejčastěji používají při vývoji webových aplikací, a to REST API, RPC API a GraphQL API. Zhodnotili jsme výhody a nevýhody jednotlivých typů webového API a já jako autor této práce jsem se rozhodl použít REST jako cechovní linii při návrhu našeho webového API.

V další kapitole se budeme zabývat technologickým zásobníkem použitých pro vývoj našeho IS v této práci.

# Technologický zásobník

V této kapitole se budeme zabývat tím, které technologie budou použity při vývoji pro náš IS a proč jsme je zvolili. Protože neexistuje požadavek na to, jaká databáze bude použita, nejprve prozkoumáme typy databází a já jako autor této práce zvolím typ databáze podle schopnosti řešit požadavky na IS, časového omezení práce, vlastních schopností a zkušeností s databází. Druhým úkolem je vytvořit API pomocí knihovny FastAPI, proto si představíme FastAPI a další nástroje, které s ním pracují. Dále si představíme React jako technologii pro vytvoření minimalistického uživatelského rozhraní pro možnost demonstrace funkčnosti, jak bylo zadáno v oddílu 1.1. Docker jsme se rozhodli představit až v posledním oddílu, protože si myslíme, že Docker má smysl pouze tehdy, když už existuje jiný software, i když první zadání této práce je o Docker.

## Typy databází

Dva požadavky, které musíme vzít v úvahu při výběru typu databáze, jsou možnost definovat proces správy a možnost definovat strukturu formuláře. Proces správy je posloupnost fází a přechodů neboli uzlů a hran z hlediska teorie grafů. Formulář obsahuje různý počet polí. Tato pole tvoří strukturu formuláře a každý formulář bude mít svou vlastní strukturu. Náš systém musí být schopen ukládat takový proces správy a takové libovolné struktury různých formulářů. Uživatelé musí být schopni inicializovat proces správy a vytvořit instanci z odpovídajícího formuláře a vyplnit do něj data. Náš systém musí být schopen uchovávat aktuální fázi instance a data vyplněná uživatelem. Pro splnění těchto požadavků budeme uvažovat dva typy moderních databází, a to relační a nerelační databázi, známou také jako SQL nebo NoSQL databáze. Poté si vybereme jeden typ databáze a její konkrétní instanci, kterou použijeme v našem systému.

### Relační databáze

Relační databázi vymyslel E. F. Codd v IBM v roce 1970 ve své výzkumné práci *A Relational Model of Data for Large Shared Data Banks* [16] . Relační databáze ukládá data do relací, které uživatel vnímá jako tabulky. Každá relace se skládá z tuplů neboli záznamů a atributů neboli polí. Fyzické pořadí záznamů nebo polí v tabulce je zcela nepodstatné a každý záznam v tabulce je identifikován polem, které obsahuje jedinečnou hodnotu (známou také jako primární klíč). Relační model rozděluje vztahy na vztahy jeden k jednomu, jeden k více a mnoho k více. Vztah mezi dvojicí tabulek je vytvořen implicitně prostřednictvím shodných hodnot společného pole (známého také jako cizí klíč). Tabulky a jejich vztahy tvoří schéma databáze. Schéma je předem definováno a často se nemění, protože změna schématu vyžaduje předefinování databáze a migraci dat. Relační databáze proto není ideální pro nestrukturovaná data a zpomaluje proces vývoje, protože schéma databáze se obvykle mění v průběhu času. Pomocí normalizace, kterou zavedl Codd ve svém prvním a druhém článku [16] [17] , však můžeme omezit redundanci, nutnost redefinovat schéma databáze (restrukturalizace), nežádoucí (anomální) operace a zároveň zachovat konzistenci ukládaných dat.

Nejběžnějším způsobem interakce s relačními databázovými systémy je použití strukturovaného dotazovacího jazyka (SQL), který umožňuje sestavovat specifické dotazy pro vytváření tabulek, vyhledávání a filtrování dat v jedné nebo více tabulkách.

Systém řízení relační databáze (RDBMS) je aplikační software, který poskytuje rozhraní API a případně uživatelské rozhraní a umožňuje nám vytvářet, udržovat, upravovat a manipulovat s relační databází.

Mezi nejoblíbenější RDBMS patří Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite, MariaDB, ...

### Nerelační databáze

Nerelační databáze, známá také jako NoSQL (lze ji chápat jako No SQL nebo Not Only SQL), je jakýkoli druh databáze, která nepoužívá koncept tabulek, polí a sloupců strukturovaných dat z relačních databází.

Existují čtyři typy databází NoSQL [12] :

#### Dokumentová databáze

Dokumentové databáze ukládají data v dokumentech, což jsou obvykle struktury podobné JSON, které podporují různé datové typy. Mezi tyto typy patří řetězce, čísla jako int, float a long, data, objekty, pole a dokonce i vnořené dokumenty. Data v dokumentu jsou uložena v párech klíč/hodnota.

Kolekce je skupina dokumentů, která obvykle uchovává dokumenty s podobným obsahem. Ne všechny dokumenty v kolekci musí mít stejná pole, protože dokumentové databáze mají flexibilní schéma.

Získávání dat z dokumentových databází se u jednotlivých dokumentových databází liší a často je proprietární. Například pro získání všech dokumentů z kolekce users:

V MongoDB má dotaz podobu řetězení metod a data lze dotazovat:

db.users.find({})

ArangoDB používá vlastní dotazovací jazyk a na data lze klást dotazy:

FOR doc IN users

RETURN doc

Ty jsou v SQL ekvivalentní jako:

SELECT \* FROM users

Mezi nejznámější dokumentové databáze nebo služby dokumentových databází patří MongoDB, ArangoDB, Amazon DynamoDB, Google Cloud Firestore, ...

#### Klíč-Hodnot databáze

Databáze klíč-hodnota (známé také jako úložiště klíč-hodnota) ukládají data ve formátu "klíč-hodnota" a jsou optimalizovány pro čtení a zápis dat. Data jsou identifikována jedinečným klíčem. Pro každý klíč existuje přesně jedna hodnota, kterou mohou být jednoduché datové typy, jako jsou řetězce a čísla, nebo složité objekty.

Tento typ databáze implementuje datovou strukturu hashovací tabulky (známou také jako slovník, mapa, hashovací mapa, tabulka symbolů), tj. klíč je zaheslován na číselnou hodnotu (hash) a hodnota je uložena na pozici definovanou hashem. Hodnota se načítá podle klíče v konstantním čase.

Úložiště klíč-hodnota nemají dotazovací jazyk jako SQL. Na hodnoty se nelze dotazovat ani v nich vyhledávat, ale pouze na klíč. Data jsou zapisována (vkládána, aktualizována a mazána) a dotazována na základě klíče.

Typickým příkladem úložiště klíč-hodnota je Redis.

#### Sloupcově orientované databáze

Sloupcově orientované databáze ukládají data po sloupcích, nikoli po řádcích jako relační databáze. V relační databázi obsahuje každý řádek hodnoty všech sloupců vztahujících se ke všem atributům instance entity. Tabulka je posloupnost řádků a každý řádek je uložen jako blok na disku. Řádek jako blok je načten se všemi svými sloupci (atributem instance). Problém nastane, pokud chceme načíst sloupec nebo malé skupiny sloupců všech řádků, musí být načten celý blok tabulky. Proto je efektivnější načítat data takovým způsobem, pokud v tabulce ukládáme data týkající se instance na jeden sloupec.

Ve sloupcově orientované databázi obsahuje každý řádek data atributu vztahující se ke všem instancím a je uložen jako blok na disku. Atribut nebo sada atributů všech instancí se načte načtením jednoho bloku nebo sady bloků bez načítání celé tabulky. Pro efektivní dotazování na data ve sloupcově orientovaných databázích je každá jednotka dat v bloku souborem dvojic identifikátor/hodnota. Sloupcově orientovaná databáze je rychlá v agregovaných operacích, například průměrný věk uživatele. Přidání atributu je rychlé také pouhým přidáním dalšího bloku na disk. Přidávání instancí je pomalé, protože každý atribut všech instancí je kontinuálně ukládán na disk.

Ve sloupcové databázi lze jako dotazovací jazyk použít SQL.

Příklady sloupcově orientovaných databází jsou Apache Cassandra, HBase.

#### Grafová databáze

Grafové databáze řeší omezení relačních databází ve schématu tím, že považují relaci za data. Data instance entity jsou uložena v uzlech a vztah je uložen v hranách. Každý uzel je kategorizován přidáním štítku, stejně jako je instance entity kategorizována v tabulce relační databáze. Uzel ukládá atributy (vlastnosti) ve formě dvojice klíč/hodnota. Hrana má také štítek, případně atributy (vlastnosti) a navíc směr. Relace se přidává prostým přidáním nové hrany bez nutnosti předefinování databáze a migrace dat.

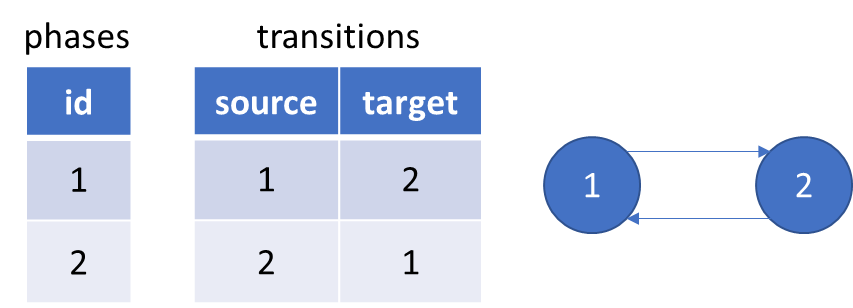
Grafová databáze je vhodná pro práci s vysoce propojenými entitami a je optimalizována pro zachycení a vyhledávání vazeb mezi datovými prvky, čímž překonává režii spojenou s JOINováním více tabulek v SQL.

Grafové databáze, jako je Neo4j, která je jednou z nejznámějších grafových databází, používají k vyhledávání dat z grafu dotazovací jazyk Cypher. Ten byl inspirován jazykem SQL a má s ním podobnost [26].

### Výběr databáze

Uvažujme první a třetí požadavek na náš IS, kterými jsou možnost definovat fáze a přechody procesu správy a možnost definovat strukturu formuláře pro tento proces.

Fáze a přechody se podobají spíše uzlům a hranám v teorii grafů. Každá fáze procesu může ukazovat na jinou fázi téhož procesu a pro uložení tohoto vztahu je vytvořen přechod. Pro různé administrativní procesy musí být různé fáze a přechody. Aby nedošlo k záměně s grafovou databází, grafová databáze se používá pro vysoce propojené entity a vztahy mezi entitami jsou náchylné ke změnám. Fáze a přechod jsou v našem systému pouze dvě entity a vztahy mezi nimi nejsou náchylné ke změně. Fáze a přechody můžeme snadno uložit do dvou tabulek, jak ukazuje na Obrázku 2. Pomocí těchto dvou tabulek můžeme snadno popsat jakýkoli proces správy, včetně přidání, změny nebo odstranění přechodu mezi dvěma fázemi. Pro splnění tohoto požadavku není třeba používat grafovou databázi. Toto téma rozšíříme ve vztahu mezi těmito dvěma tabulkami v kapitole 4.



Obrázek 2 - Relační databáze pro uložení fází a přechodů

Zdroj: vlastní

Formulář je strukturou pro jeho instanci. Struktura formuláře určuje, kolik polí se v instanci zobrazí a kolik jich může uživatel vyplnit. Instanci lze vytvořit z formuláře a uživatel do ní může vyplňovat údaje. Z pohledu programátora je formulář jako třída a instance je jako objekt, který je z této třídy vytvořen. Problém je v tom, že struktura formuláře není programátorem předem definována jako tabulka v databázi pro ukládání dat vyplněných uživatelem, protože tato struktura je sama o sobě daty a je definována uživatelem a my ji musíme uložit. Když pak jiný uživatel, například žadatel, chce instanci vytvořit a vyplnit do ní data, musíme tato data uložit tak, abychom je mohli načíst a abychom věděli, do kterého pole patří.

Jedním z možných řešení je, že tuto strukturu formuláře uložíme do dokumentové databáze, v níž kolekce s názvem "forms" bude uchovávat veškerou strukturu formuláře v dokumentech s formátem podobným JSON. Tyto dokumenty mohou mít různá pole vzhledem k povaze flexibilního schématu dokumentové databáze. Klíčem může být název pole a hodnotou mohou být jakákoli metadata popisující toto pole v libovolném datovém formátu. Instance budou uloženy v dokumentech jiné kolekce s názvem "instance", v níž může být název pole odvozen z jednoho dokumentu kolekce "formulář" jako klíč a hodnota pole je údaj od uživatele, metadata pole lze použít k ověření této hodnoty.

Ale také, jak již bylo řečeno, můžeme rozdělit strukturu formuláře a data vyplněná uživatelem a uložit je odděleně do různých tabulek jako běžná data. Tohoto řešení lze dosáhnout pomocí relační databáze a normalizace. O tomto řešení budeme hovořit v kapitole 4.

Důvodem pro volbu relační databáze při vývoji našeho informačního systému je učení se něčemu zcela novému, včetně způsobu návrhu a nástroje, což je překážkou pro přijetí. Zvládnutí nástroje se správnou metodou návrhu vyžaduje čas. Proto s ohledem na schopnost relační databáze řešit požadavky na IS, časové omezení práce a vlastní schopnosti a zkušenosti s relační databází jsem se jako autor této práce rozhodl zvolit pro vývoj našeho IS relační databázi. RDBMS, kterou jsem se rozhodl v této práci použít, je PostgreSQL.

## FastAPI

### FastAPI

Jak je uvedeno v dokumentaci [FastAPI](https://fastapi.tiangolo.com/), jedná se o moderní, rychlý (vysoce výkonný) webový framework pro vytváření API v jazyce Python 3.6+ založený na standardních typových nápovědách Pythonu.

FastAPI vychází z OpenAPI a je s ním plně kompatibilní. Automaticky generuje specifikaci pro všechna definovaná API odpovídající standardu OpenAPI. Specifikace obsahuje cesty, parametry, které přebírají, atd. Soubor pro tuto specifikaci je v datovém formátu JSON (známém také jako schéma JSON) s názvem openapi.json. Tento soubor se používá pro interaktivní dokumentační systém, jako je Swagger UI, k vizualizaci a interakci se zdroji API.

Graphical user interface, text, application

Description automatically generated

Obrázek 3 - Interaktivní dokumentační systém automaticky vytvořený pomocí Swagger UI z openapi.json

Zdroj: vlastní

### SQLAlchemy

K implementaci webového rozhraní API se používá jazyk Python. Nejznámějším databázovým ovladačem Pythonu pro PostgreSQL je psycopg2. Pro zjednodušení procesu vývoje však používáme SQLAlchemy, sadu nástrojů SQL v jazyce Python a objektově relační mapovač (ORM). ORM lze použít k vytvoření nebo odstranění tabulky jako DDL a používá se hlavně k vkládání, načítání, aktualizaci, mazání dat jako DML, ale s vyšší úrovní abstrakce. Pomocí ORM můžeme zpracovávat data v paradigmatu OOP v jazyce Python a bez použití jazyka SQL (i když použití SQLAlchemy ORM stále vyžaduje znalosti jazyka SQL).

### Swagger specifikace a OpenAPI specifikace

Specifikace OpenAPI byla původně založena na specifikaci Swagger (verze 1.0 až verze 2.0), kterou poskytla společnost SmartBear Software. V roce 2015 Linux Foundation oznámila Open API Initiative, která specifikaci Swagger 2.0 rozšířila a přejmenovala na specifikaci OpenAPI [11] .

Specifikace Swagger a specifikace OpenAPI se používají k popisu a dokumentaci rozhraní RESTful API. Definici OpenAPI pak mohou používat nástroje pro generování dokumentace k zobrazení rozhraní API, nástroje pro generování kódu ke generování serverů a klientů v různých programovacích jazycích, testovací nástroje a mnoho dalších případů použití.

Příkladem nástroje pro generování dokumentace je Swagger UI, který se používá ve FastAPI.

## React

Dalším úkolem této práce je vytvořit minimalistické uživatelské rozhraní pro možnost demonstrace funkčnosti. Pro splnění tohoto úkolu použijeme React a CytoscapeJS.

React je knihovna jazyka JavaScript (JS) pro vytváření uživatelských rozhraní. Lze ji použít k vytvoření jednostránkové webové aplikace, která umožňuje uživateli vyžádat si webovou stránku (HTML, JS, CSS, ...) pouze jednou a tato stránka si pomocí kódu JS vyžádá data z rozhraní API a aktualizuje strom DOM. Vytvoří virtuální DOM sloužící k porovnávání a aktualizaci stromu DOM, který bude vykreslen do prohlížeče. Proces porovnávání a aktualizace stromu DOM se nazývá rekonciliace. React implementuje heuristický algoritmus porovnávání založený na dvou předpokladech:

* Dva prvky různých typů vytvoří různé stromy.
* Podřízené prvky mohou být stabilní napříč různými vykresleními s klíčovou rekvizitou.

Tyto dva předpoklady pomáhají systému React snížit časovou složitost z O(n3) na O(n) ve srovnání s obecnými řešeními pro generování minimálního počtu operací pro transformaci jednoho stromu na jiný [15] .

Dalším hlediskem při vytváření uživatelského rozhraní našeho systému je vizualizace procesu správy pro uživatele, kterou budeme realizovat pomocí CytoscapeJS. Jedná se o knihovnu napsanou v čistém JS, která nám umožňuje vizualizovat naše fáze a přechody pro proces správy jako směrovaný graf uzlů a hran, jak ukazuje Obrázek 4.

Chart

Description automatically generated with medium confidence

Obrázek 4 - Vizualiace fází a přechodů pomocí CytoscapeJS

## Docker

### Docker a kontejner

Docker je virtualizační software na úrovni operačního systému (často se označuje jako kontejnerizace), který balí aplikaci a její závislosti do softwarové jednotky, tzv. kontejneru. Snižuje plýtvání zdroji, např. procesorem, pamětí RAM, úložištěm, licencí OS pro každý aplikační software pomocí virtualizační technologie tím, že používá sdílený hostitelský OS pro všechny kontejnery na něm spuštěné. Poskytuje také způsob rychlejšího sestavení, testování a nasazení bez bariéry kompatibility platforem [13] [14].

Chart, treemap chart

Description automatically generatedGraphical user interface, application

Description automatically generated

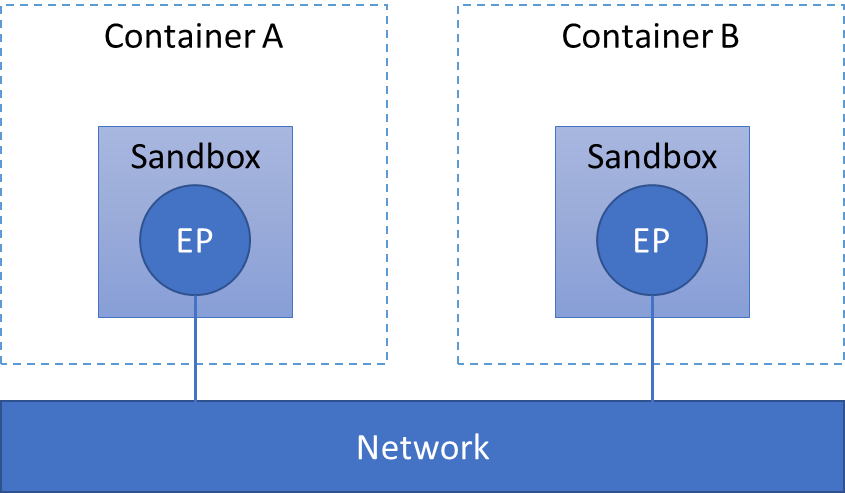
Obrázek 5 - Kontejnerizace vs. Virtualizace

Zdroj: <https://www.docker.com/resources/what-container>

Kontejner je standardní softwarová jednotka, která balí kód a všechny jeho závislosti, takže aplikace běží rychle a spolehlivě z jednoho výpočetního prostředí do druhého. Obraz kontejneru (zkráceně image) je lehký, samostatný, spustitelný balíček softwaru, který obsahuje kód, běhové prostředí, systémové nástroje, systémové knihovny a nastavení, které jsou potřebné ke spuštění aplikace. Obrazy se stávají kontejnery, když jsou spuštěny na Docker Engine za běhu.

### Síťování v Dockeru a Docker Compose

Návrh sítě Docker je založen na Container Network Model (CNM), který obsahuje 3 hlavní komponenty: Sandbox, Endpoint and Network [13] .



Obrázek 6 - Docker network záložen na CNM

Zdroj: vlastní

* A Sandbox obsahuje konfiguraci síťového zásobníku kontejneru. To zahrnuje správu rozhraní kontejneru, směrovací tabulky a nastavení DNS. Sandbox může obsahovat mnoho koncových bodů z různých sítí.
* An Endpoint připojuje Sandbox k Network jako virtuální síťové rozhraní (např. veth). Endpoint může patřit pouze do jedné sítě a může patřit pouze do jednoho Sandboxu, pokud je připojen.
* A Network je softwarová implementace přepínače, která seskupuje koncové body a umožňuje jim vzájemnou přímou komunikaci.

CNM je implementován pomocí libnetwork a ovladače. Libnetwork poskytuje API pro uživatele, jako je Docker nebo třetí strana, k vytváření a správě objektů CNM, například objektů Network nebo Endpoint. Vlastní implementace vytváření a správy těchto objektů uživatelem, jako je Docker nebo třetí strana, se nazývá ovladač. Ve výchozím nastavení Dockeru existuje několik ovladačů, které poskytují základní síťové funkce, např. bridge, host, overlay, ipvlan, macvlan. Další zásuvné moduly síťových ovladačů jsou poskytovány třetími stranami v Docker Hubu.

Když spustíme Docker, automaticky se vytvoří výchozí bridge network a nově spuštěné kontejnery se k němu připojí. Kontejnery spuštěné na stejném hostiteli mohou mezi sebou komunikovat ve výchozí bridge network pouze pomocí IP adres (nikoli podle názvu kontejneru). User‑defined bridge jinak zajišťují automatické rozlišení DNS mezi kontejnery a umožňují kontejnerům v síti komunikovat podle názvu kontejneru.

Docker Compose (zkráceně Compose) je nástroj pro definování a spouštění vícekontejnerových aplikací Docker jako služeb, které spolu komunikují. Ve výchozím nastavení Compose nastaví v jednom hostiteli bridge network pro naše kontejnery a dynamicky přidělí IP adresu pro každý kontejner. Každý kontejner se připojí k výchozí síti a je dosažitelný ostatními kontejnery v této síti podle názvu kontejneru definovaného v souboru Compose (obvykle ve formátu YAML). IP adresa je každému kontejneru automaticky přidělena ve výchozím nastavení v rozsahu 172.17.0.0/16. Pokud provedeme změnu konfigurace služby a spustíme docker‑compose up pro její aktualizaci, starý kontejner se odstraní a nový se připojí k síti pod jinou IP adresou, ale stejným názvem. Proto je efektivnější používat při komunikaci mezi službami ve výchozí síti pro vývoj nebo produkci název kontejneru než se snažit konfigurovat, jakou IP adresu má kontejner.

Všimněte si, že bridge network je soukromá síť omezená na jednoho hostitele Docker a všechny kontejnery v této síti mohou mezi sebou komunikovat pouze prostřednictvím mostu (virtuálního přepínače) vytvořeného ovladačem bridge. Pro příjem a vysílání provozu mimo tuto síť musíme použít mapování portů. Docker umožňuje namapovat port kontejneru na port hostitele. Veškerý provoz s nakonfigurovaným portem přicházející na hostitele bude směrován na kontejner a umožní kontejneru komunikovat s okolním světem.

Náš informační systém je sada spolupracujících kontejnerů využívajících Docker Compose jako první zadání této práce. Při vývoji našeho IS budeme používat Docker a Docker Compose. Strukturu spolupracujících kontejnerů představíme v následující kapitole poté, co představíme všechny ostatní služby.

## Git a GitHub

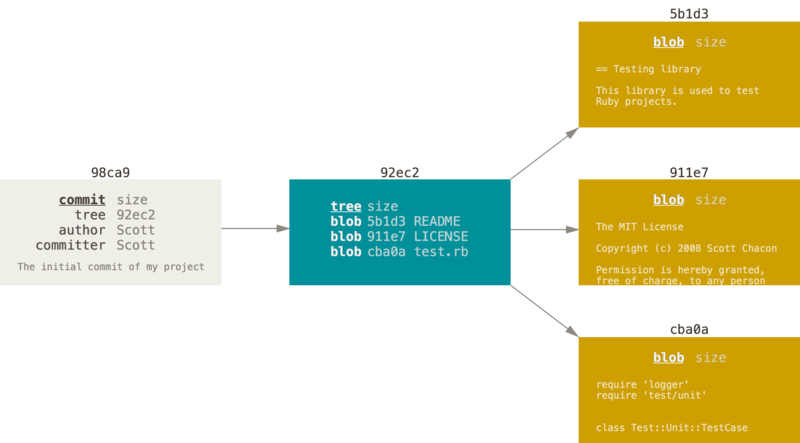
Git je distribuovaný systém pro správu verzí (Version Control System, VCS), který je navržen pro rychlou a efektivní správu malých i velmi rozsáhlých projektů. Systém Git byl vyvinut komunitou vývojářů Linuxu (a zejména Linusem Torvaldsem, tvůrcem Linuxu) při vývoji jádra Linuxu v roce 2005 [19] .

GitHub je platforma pro hostování zdrojového kódu pro správu verzí a spolupráci. Umožňuje nám vyvíjet projekt s ostatními vývojáři odkudkoli. Nabízí distribuovanou správu verzí a správu zdrojového kódu jako Git, a navíc vlastní funkce.

Kromě účelu systémů Git a GitHub, které slouží ke správě verzí našeho zdrojového kódu, vidíme v systému Git inspirativní myšlenku, kterou lze aplikovat v našem systému a která byla představena v následujícím pododdílu.

### Inspirace systémem Git

Systém Git o svých datech uvažuje spíše jako o sérii snímků miniaturního souborového systému. Pro každý soubor v našem projektu uloží systém Git jeho obsah do jiného souboru ve své vlastní databázi (ve složce .git/object) s hashem pomocí SHA-1 jako název souboru. Systém Git nazývá tento typ souboru objektem blob. Git používá objekt tree uložení buď názvů souborů a hashů skupiny blobů, nebo názvu adresáře a hashe jiného(jiných) tree. Objekt tree je jako adresář v souborovém systému UNIX a blob jsou jako soubor. Při commitu systém Git vygeneruje ze struktury souborů našeho projektu top-level tree, což je "snímek" našeho projektu. Porovnává hash jednotlivých souborů, a pokud se soubory nezměnily, nezmění se ani hash a Git neukládá znovu odpovídající blob, pouze hash k předchozímu identickému blobu, který již uložil. Git pak použije objekt commit k uložení hashe top-level tree, hashe předchozí commit, pokud existuje, informace o autorovi s časovým razítkem a zprávu o commitu. Na obrázku 6 je příklad blobu, tree, commit z knihy *Pro Git*, kterou napsali Scott Chacon, Ben Straub.

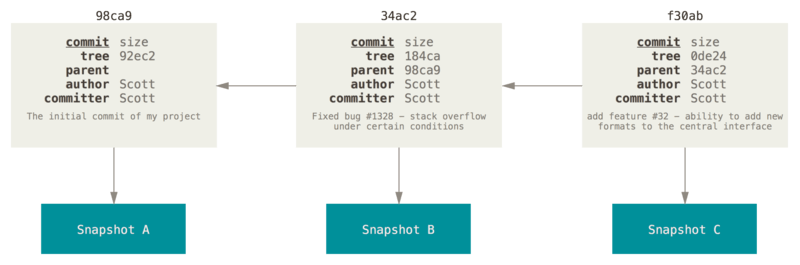


Obrázek 7 - Vizualizace blob, tree a commit

Zdroj: [19]

Jiný přístup odlišný od systému Git spočívá v ukládání založených souborů a rozdílů v jejich obsahu v průběhu času (často se označuje jako delta-based version control). Při tomto přístupu, pokud chceme získat soubor v určitém commitu, musíme počítat ze založeného souboru a přičíst všechny jeho delty až do zadaném commitu. Tento přístup vyžaduje více výpočtů a složitější mechanismus větvení a slučování v porovnání se systémem Git, který má lehké, rychlé a velmi jednoduché operace větvení a slučování [19] .

Dále se každé commit propojí s předchozím commitem prostřednictvím hashe předchozího commitu a vytvoří se historie commitu. Pokud nebudeme brát v úvahu mechanismus větvení a slučování commitu poskytovaný systémem Git, bude historie commitu lineární a každé commit můžeme považovat za chronologický záznam našeho projektu a můžeme projekt rekonstruovat a prozkoumat při kterémkoli commitu v historii commitu.



Obrázek 8- Vizualizace historie commitů

Zdroj: [19]

Pokud uvažujeme o auditu žádosti žadatele, jak jsme jej představili v pododdílu 1.3, potřebujeme pouze lineární historii žádosti prostřednictvím posloupnosti fází, protože historii žádosti nelze větvit a slučovat jako zdrojový kód projektu, kde spolupracuje více vývojářů a upravují zdrojový kód paralelně. Žádost se může objevit pouze v jedné fázi procesu jako v administrativním procesu s papírem, kde jeden list papíru nemůže existovat ve dvou nebo více fázích. Bez ohledu na mechanismus větvení a slučování commitu poskytovaný systémem Git používáme tento přístup k řešení požadavků v této práci z důvodu jeho jednoduchosti a robustnosti. Tuto myšlenku použijeme jednodušším způsobem a upravíme ji tak, abychom dosáhli cíle našeho systému. Podrobnější popis naší úpravy je uveden v oddílu 4.2.4.

## Shrnutí

V této kapitole jsme představili 2 typy databází, a to relační a nerelační. Každý typ má své výhody a nevýhody a jeho použití závisí na konkrétním případu použití. Já jako autor této práce jsem se rozhodl zvolit pro ukládání dat jako systém řízení databází PostgreSQL, a to z důvodu jeho schopnosti řešit požadavky na IS, časového omezení práce a vlastních schopností a zkušeností s relační databází. Pro vývoj našeho IS jsme zavedli i další frameworky a knihovny, kterými jsou FastAPI pro tvorbu API, React pro tvorbu uživatelského rozhraní, Docker jako kontejnerizační technologie, Git jako systém pro správu verzí a GitHub jako platforma pro hostování zdrojových kódů, jak je uvedeno v zadání této práce. V další kapitole představíme návrh našeho IS a způsob implementace našeho IS pomocí těchto technologií.

# Diskuse a implementace

V této kapitole si ukážeme, jakým způsobem implementujeme pokyny popsané na začátku této práce, abychom splnili požadavky na informační systém, získali nezbytné výstupy a dosáhli cíle práce.

V první části se budeme podrobně zabývat procesem administrace. Odpovíme na následující otázky: Jak definovat posloupnost fází administrativního procesu? Jak definovat, kdo bude zodpovědný za jednotlivé fáze? Co je to struktura formuláře a jak ji definovat v administrativním procesu? Jak žadatel vytvoří instanci z definovaného formuláře a tranzituje ji fázemi? Co je předmětem auditu? Jaký je účel auditu? Kdy by se měl vytvořit auditní stopu? Abychom na tuto otázku odpověděli, rozebereme si několik příkladů procesu správy, zobecníme pojmy a vyvodíme charakteristické rysy těchto příkladů.

Tyto charakteristiky použijeme v datovém modelu našeho systému a uložení omezení pro tento model, které popíšeme v oddílu 4.2.

Oddíl 4.3 bude popisovat návrh rozhraní API a jeho implementaci pomocí FastAPI. Popíšeme také vzor API a způsob, jakým tento vzor v architektuře API aplikujeme.

Technickou dokumentaci popíšeme v oddílu 4.4, ve kterém rozdělíme dokumentaci na nízkoúrovňovou a vysokoúrovňovou.

Minimalistické uživatelské rozhraní vytvořené pomocí Reactu v oddíle 4.5 ukáže, jak vytvořit fázi s určenou pozicí a jak by měla vypadat instance, přechod instance a historie commitů instance jako ukázka funkcí API.

V oddíle 4.6 bude představena sada kontejnerů vytvořená pomocí nástroje Docker Compose v podobě bridge network kontejnerů na jednom hostiteli a komunikace mezi nimi.

Nakonec je v oddíle 4.7 popsán potřebný výstup pro tuto práci.

## Administrativní proces

V organizaci je proces správy definován předpisem jako sled fází a obvykle je doprovázen formulářem. Žadatel z formuláře vytvoří instanci, zahájí první fázi procesu a odešle ji do dalších fází (případ, kdy se v různých fázích používají různé formuláře, se považuje za rozdělení na různé administrativní procesy, a proto se jím v této práci dále nezabýváme). Přání žadatele vyřešit vlastní záležitost tím, že vytvoří instanci, zahájí proces a projde fázemi s cílem dosáhnout svého přání, se nazývá **žádost** (**request**). Termín žádost se používá také pro označení instance s konkrétní posloupností fází, kterými instance prošla.

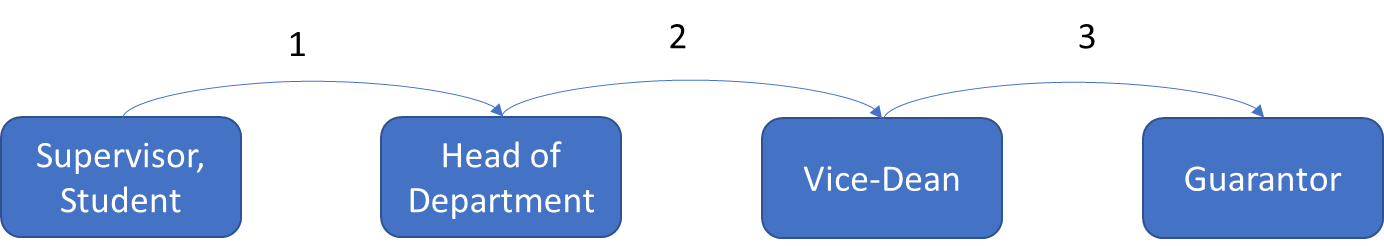
Odpovědnosti za různé fáze procesu jsou přiděleny různým subjektům, např. oboru, fakultě, katedře, pracovní skupině, studijní skupině... my tento termín zobecňujeme a nazýváme je **skupinou** (**group**). Tyto odpovědnosti jsou dále specifikovány a vyjádřeny jako **role** (**role**) ve skupině. Role ve skupinách mají často stejný vzor, pokud mají skupiny stejnou úroveň v organizační struktuře. Například na univerzitě bude mít každá fakulta děkana a proděkany, každá katedra bude mít vedoucího katedry, profesory, docenty, asistenty, .... Může nastat případ, kdy skupiny na stejné úrovni v organizační struktuře nemají stejnou roli, například místo vedoucího katedry je zástupce vedoucího katedry. V obou případech pro specifikaci role vztahující se ke skupině používáme termín **pozice (position)**, např. rektor Univerzity Obrany, děkan Fakulty Vojenských Technologií nebo vedoucí Katedry Informatiky a Kybernetických Operací ... jsou všechno pozice. Pozice také může představovat obecnou roli a není vztažena k žádné skupině. V tomto případě jsou pozice a role rovnocenné a oba pojmy se používají zaměnitelně.

### Fáze v administrativním procesem

**Fáze** (**phase**) je samostatná část procesu, ve které uživatel(é) s přidělenou pozicí (pozicemi) zpracovává(jí) sekci (sekce) instance a ve které se vytváří auditní stopa. Osoba, která zpracovává instanci ve fázi, je **zpracovatelem** (**handler**) této fáze. V jedné fázi může být více zpracovatelů. "Zpracovavat" (handle) znamená, že instance bude v některé části upravena a/nebo mohou být provedeny některé akce v reálném životě, např. kontrola platnosti informací vyplněných žadatelem.

Proces bude začínat jednou fází a končit jednou fází, nazýváme ji **počáteční** a **koncová fáze** (**begin phase** a **end phase**). Další fáze mezi těmito dvěma fázemi nazýváme **tranzitní fáze** (**transit phase**). Proces může začínat dvěma nebo více zpracovateli v jedné fázi, ale nemůže začínat více fázemi. Pokud je proces navržen tak, aby začal jednou počáteční fází z více počátečních fází (nebo skončil jednou koncovou fází z více koncových fází), měl by se rozdělit na samostatné procesy. Pokud je proces navržen tak, aby začal s více počátečními fázemi (nebo skončil s více koncovými fázemi) paralelně, měl by být sloučen do jedné počáteční fáze (nebo koncové fáze) s více zpracovateli. Uvažujme příklad, kdy administrativní proces definovaný ve STUDIJNÍM ROZKAZ DĚKANA FAKULTY VOJENSKÝCH TECHNOLOGIÍ, Zadávání a zpracování závěrečných prací, článek 5 Zpracování zadání závěrečné práce a jeho schválení [18]:

* Vedoucí práce společně se studentem zpracují Zadání závěrečné práce na předepsaném formuláři. Studentem a vedoucím práce podepsané Zadání závěrečné práce předá student vedoucímu katedry
* Vedoucí katedry se vyjádří k zadání. Zadání závěrečných prací vedoucí katedry předá PdSPČ.
* PdSPČ předkládá všechna Zadání závěrečných prací studentů fakulty garantovi, který písemně rozhodne o stanoveném Zadání závěrečných prací.
* Po schválení Zadání závěrečných prací Studijní skupina děkanátu fakulty každé Zadání závěrečné práce opatří úředním razítkem. Následně jsou dva výtisky Zadání závěrečné práce předány studentovi, jeden výtisk obdrží vedoucí závěrečné práce a jeden výtisk Studijní skupina děkanátu fakulty založí do spisu studenta.



Obrázek 9 - Lineární administrativní process

Čislo indikuje poradí přechodu

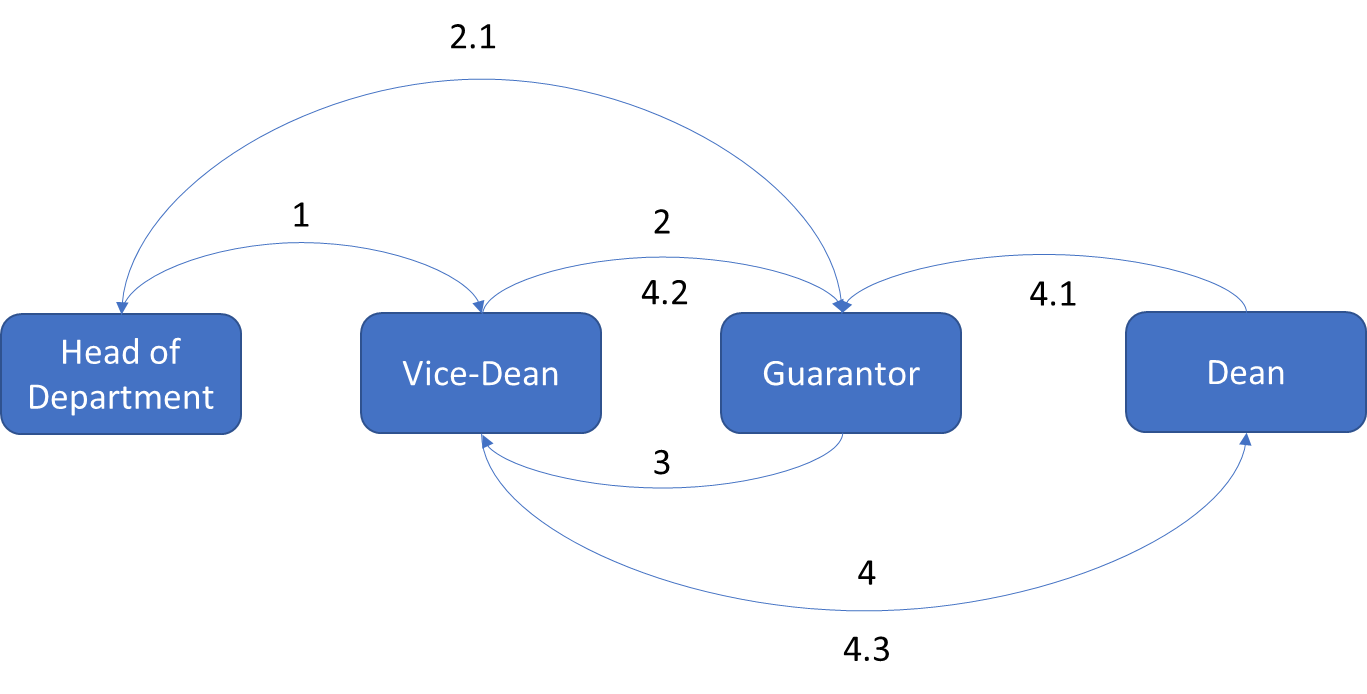
Zdroj: vlastní

Vidíme, že existuje pouze jedna počáteční fáze, kterou řeší vedoucí práce a student. Jejím účelem je připravit zadání pro závěrečnou práci. Další tranzitní fázi zpracovává vedoucí katedry a jejím účelem je dosáhnout vyjádření vedoucího katedry k zadání. Poté instance přechází do další fáze, kterou zpracovává proděkan, a proděkan shromáždí všechny instance Zadání závěrečné práce garantovi. Pouze jednu koncovou fázi vyřizuje garant, který písemně rozhodne o všech instancích. Posledním bodem je pouze předání kopie instance účastníkům procesu, instance se dále neupravuje, a proto již další fáze neprobíhá a instance končí ve fázi Garanta.

Výše uvedený příklad ukazuje pouze lineární proces, kdy instance přechází do další fáze bez možnosti návratu zpět. V důsledku toho nepopisuje všechny možné přechody v praxi. Například pokud vedoucí katedry zjistí překlep studenta, neexistuje žádný přechod zpět do první fáze. To, že přechod není popsán, však neznamenalo, že jej nelze provést. Vedoucí katedry může studenta o překlepu informovat a požadovat, aby student případ opravil a znovu odeslal. V tomto případě je přechod implicitně proveden vedoucí katedry, a protože nemá velký vliv na proces, není v předpisu popsán.

Další proces je definován v článku 8 [18] , oponent závěrečné práce, který popisuje přechod zpět takto:

* Vedoucí katedry odevzdává návrh oponentů závěrečných prací PdSPČ (1).
* PdSPČ si vyžádá stanovisko garanta k oponentům závěrečných prací (2).
* V případě neschválení navrženého oponenta závěrečné práce garant studijního programu zdůvodní neschválení a navrhne nového oponenta závěrečné práce. Návrh na nového oponenta závěrečné práce může garant studijního programu konzultovat s vedoucím katedry (2.1).
* Stanovisko garanta k navrženému seznamu oponentů závěrečných prací je předáno děkanovi prostřednictvím PdSPČ (3) (4). Děkan schválí seznam navržených oponentů závěrečných prací. V případě neschválení navrženého oponenta závěrečné práce zdůvodní neschválení a navrhne nového oponenta závěrečné práce. Nový návrh oponenta závěrečné práce se předá garantovi studijního programu ke schválení (4.1). Garant studijního programu své stanovisko předá děkanovi prostřednictvím PdSPČ (4.2) (4.3). Děkan nového oponenta závěrečné práce schválí.
* Seznam obhajovaných závěrečných prací včetně vedoucích a oponentů závěrečných prací je zveřejňován ve studijním rozkaze děkana.



Obrázek 10 - Nelineární administrativní proces

Zdroj: vlastní

Vidíme, že přechody (2.1), (4.1), (4.2) a (4.3) jsou přechody zpět, když zpracovatelé neschválí informace v instanci. Požadavek vytvořený z tohoto procesu může projít tímto přechodem, ale není to nutné.

Vidíme také, že přechody (2) a (4.2), (4) a (4.3) měly být odlišnými přechody, protože požadavek jimi prochází s jiným účelem. Přechod by však měl pouze definovat tuto dostupnost, zda požadavek může přejít z jedné fáze do druhé, bez ohledu na účel přechodu. Účel přechodu může být různý v závislosti na požadavku, a proto by se měl nechat určit účastníky požadavku. Definování více přechodů z jedné fáze do druhé samostatně je nadbytečné a navrhovatel procesu nemusí zachytit všechny případy, ke kterým při přechodu mezi těmito dvěma fázemi došlo.

Všimněte si, že neexistuje žádný přechod z jedné fáze do sebe sama. V praxi se proces dělí na fáze, protože: za prvé, každou fázi bude zpracovávat jiný zpracovatel na jiném pracovišti; za druhé, dokončení jedné fáze je požadavkem na zpracování další fáze. Pokud je tedy proces navržen s fází, která přechází sama na sebe, měla by to být pouze jedna fáze. Pokud navrhovatel zamýšlí vytvořit proces takovým způsobem pro účely auditu, měl by být proces rozdělen na dvě fáze.

### Určená pozice a ředitel

V obou výše uvedených příkladech může každou fázi zpracovávat více zpracovatelů, ale přechod může provést pouze jedna pozice. Taková pozice se nazývá **určená pozice** a zpracovatel s určenou pozicí se nazývá **ředitel** fáze, protože ředitel řídí směr instance v procesu. Například ředitelem počáteční fáze v prvním příkladu je student; řediteli všech ostatních fází jsou zpracovatelé těchto fází.

Určená pozice nemusí být jednou z přidělených pozic fáze a ředitel v takovém případě provádí pouze přesměrování, nikoliv modifikaci instance. Všimněte si, že v obou výše uvedených příkladech proděkan nemodifikuje žádné pole v instanci, ale provádí dvě akce: 1 je kontrola platnosti informací v instancích, 2 je přesměrování instance do další fáze. V prvním příkladu nedochází k přímému přechodu z vedoucího katedry na garanta. Ve druhém příkladu není přímý přechod z vedoucího katedry na garanta ani z garanta na děkana. Pokud v obou příkladech sloučíme fázi proděkana s fází vedoucího katedry, v druhém příkladu sloučíme fázi proděkana s fází garanta, můžeme proděkana označit za pouhého ředitele, nikoliv za zpracovatele těchto fází. Fáze s ředitelem a garantem z výše uvedeného příkladu lze vizualizovat na Obrázku 11.

Graphical user interface

Description automatically generated

Obrázek 11 - Administrativní proces s řediteli

Zdroj: vlastní

### Struktura formuláře

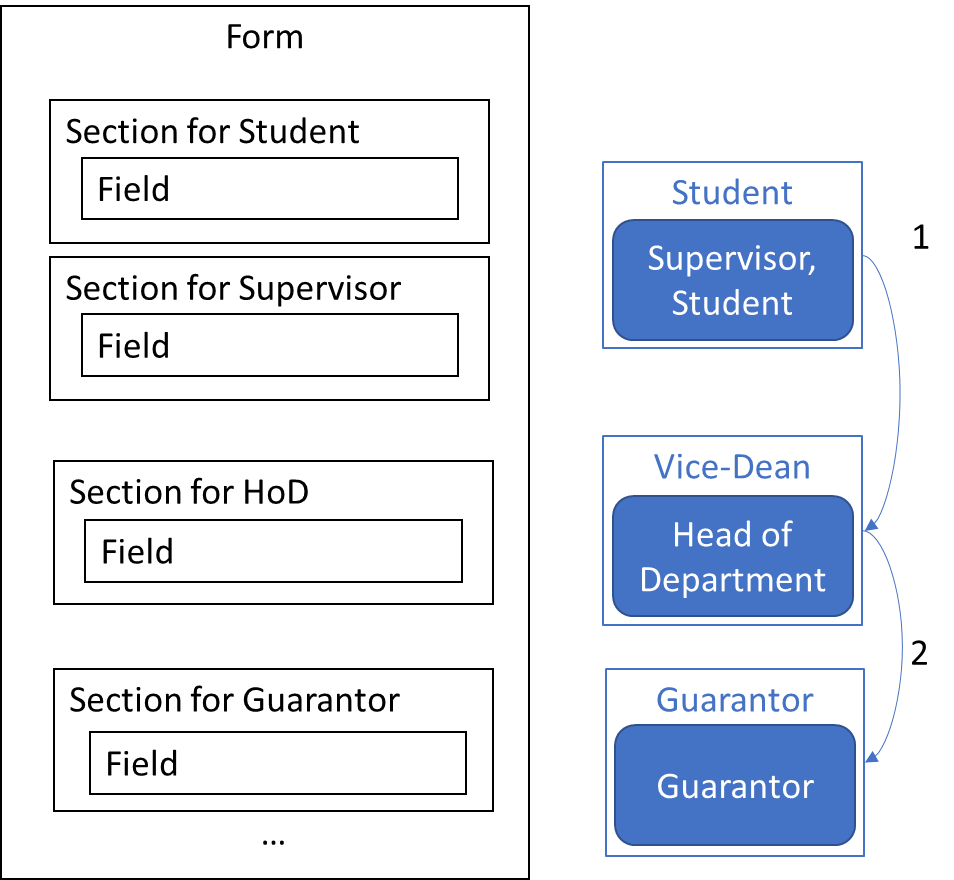
**Formulář** definovaný pro administrativní proces musí obsahovat **pole** a každé pole je přiřazeno pouze jedné pozici. Údaje vyplněné v poli může ovlivnit více zpracovatelů, ale pole vyplňuje pouze jeden zpracovatel, například vedoucí závěrečné práce a student mohou vypracovat Zadání závěrečné práce, avšak ve formuláři zadání vyplňuje pouze vedoucí, nikoliv student. Pro zjednodušení struktury formuláře můžeme pole (pole), která jsou přiřazena k pozici, seskupit do **sekce**. Jinými slovy sekce poskytují funkční metodu seskupení formuláře.

Sekce také poskytuje sémantickou metodu při seskupování polí. Obsluha bude vědět, že pole, která musí zpracovat, jsou v jedné sekci, a ne v několika různých sekcích. Sekce se proto používá ke strukturování formuláře a pomáhá zpracovatelům vědět, která pole musí zpracovat a která ne.

Prezentace polí přiřazených k jedné pozici jsou obvykle uspořádány na jedno místo ve formuláři a jen zřídka na samostatná místa a tato pole můžeme také seskupit do jedné sekce. Sekce tedy poskytuje prezentační metodu při seskupování polí. Pořadí prezentace sekcí ve formuláři bude obvykle odpovídat pořadí fáze odpovídajícího procesu, kterým musí instance projít. Formulář je však lineární dokument a pořadí sekce může odpovídat pořadí fáze pouze tehdy, je-li proces rovněž lineární. V případě nelineárního procesu pořadí sekcí nemůže odpovídat pořadí fází. Tvůrce administrativního procesu a formuláře obvykle tento problém kompenzuje tím, že se snaží definovat proces a prezentovat sekce ve formuláři co nejlineárněji.

Pro každou fázi existuje více sekcí, které lze zpracovat, a každá sekce může obsahovat více polí. V praxi každé pole v platné instanci vyplňuje pouze osoba s přidělenou pozicí. Strukturu formuláře si lze představit na obrázku níže.

Pro každou fázi existuje více sekcí, které lze zpracovat, a každá sekce může obsahovat více polí. V praxi pole sekce vyplňuje pouze osoba s přidělenou pozicí. Strukturu formuláře si lze představit na Obrázku 12.



Obrázek 12 - Struktura formuláře

Zdroj: vlastní

### Instance a její přechod

Chce-li vytvořit žádost, žadatel vytvoří instanci z formuláře, který je definován pro proces správy korespondence v nařízení, a zahájí proces. Po vyplnění všech polí určených pro žadatele přejde instance do další fáze procesu a mohou nastat dva případy: žadatel ví, nebo neví, kdo jsou zpracovatelé další fáze.

Pokud žadatel patří k organizaci, měl by vědět a musí vědět, kdo jsou zpracovatelé další fáze. Například student musí znát vedoucího katedry, ke které patří, nebo vedoucí katedry musí znát proděkana fakulty, ke které jeho katedra patří. Takovému typu administrativního procesu říkáme vnitřní proces.

Pokud žadatel nepatří do organizace, ale je subjektem mimo organizaci, nemusí vědět, kdo budou zpracovatelé dalších fází. Například nový student, který se chce zapsat na univerzitu, neví, kdo bude příjemcem jeho žádosti. Tento typ administrativního procesu nazýváme externí proces. Je to typický typ administrativního procesu ve veřejném sektoru, kde organizace řeší záležitosti občanů. Žadatel buď pošle žádost zástupci skupiny, která je určena k zpracování takového typu žádostí, nebo se dostaví do organizace a žádost přímo předá pracovníkovi, který ji zpracuje. Organizace ve veřejném sektoru navíc bude mít více zpracovatelů se stejnou pozicí, obvykle zaměstnanců, aby mohla vyřídit více žádostí najednou. Žádosti se zařadí do fronty a zpracovatel není určen, dokud žádost nepřijme. Koordinátor, buď systém, nebo osoba, pak bude koordinovat zpracovatele pro zpracování případů.

Informační systém v této práci dokáže tyto dva případy zpracovat. Jediným společným požadavkem pro tyto dva případy je to, že žadatel musí určit, kdo je ředitelem další fáze. V prvním případě může žadatel určit, kdo je příjemcem dalších sekcí. Ve druhém případě může žadatel ponechat příjemce dalších sekcí jako nespecifikované a při přechodu instance do další fáze určí příjemce sekcí ředitel další fáze. Ředitel v druhém případě hraje roli zástupce skupiny a roli koordinátora.

Ředitel může nejen specifikovat příjemce pro úseky, ale musí mít také možnost změnit příjemce úseku. Například: Vedoucí oddělení vybral vedoucího práce pro studenta, ale vedoucí práce to odmítl, takže vedoucí oddělení musí pro sekci vedoucího práce vybrat jiného příjemce.

Po přijmutí instance se příjemce stává zpracovatelem sekce a může upravovat pole, která jsou mu přidělena. Ředitel aktuální fáze může instanci kdykoli tranzitovat do další fáze a proces bude pokračovat až do koncové fáze.

V koncové fázi, kdy je instance kompletně zpracována všemi zpracovateli, může ředitel označit instanci za kompletně zpracovanou a instanci již nelze dále upravovat.

### Auditování and auditní stopa

Předmět auditu může být různý, záleží na tom, jaký je účel auditu. Můžeme auditovat přihlášení uživatelů do systému, akce uživatele v systému, jako je vytvoření skupiny, změna role, odstranění pozice, vytvoření formuláře... pro účely zabezpečení. Nebo můžeme auditovat přistoupení a odchod uživatelů do skupiny, přidělení a zrušení pozice pro uživatele, zveřejnění a zrušení formuláře... Pro podporu procesu správy je však nejdůležitějším předmětem, který musíme auditovat, požadavek žadatele.

Veškeré úsilí žadatele v žádosti spočívá v tom, aby v každé fázi dosáhl povolení, pokynu, souhlasu, schválení, vyjádření, stanoviska nebo názoru na svou žádost od zpracovatelů a dosáhl konce procesu. Můžeme říci, že žadatel chce dosáhnout svého cíle a zpracovatelé mu dosažení cíle schvalují. Z hlediska administrativy je koncovou fází, kde je žádost vyhodnocena na nejvyšší úrovni administrativy. Zpracovatel v koncové fázi obvykle neví, kdo žádost editoval, v jakém čase byla žádost editována, jakými fázemi žádost prošla a jaký byl v té době obsah žádosti. Tyto informace jsou rozhodující, když koncový zpracovatel vyhodnocuje žádost a rozhoduje o schválení dosažení cíle žadateli. Například při rozhodování o oponentovi závěrečné práce chce děkan vědět, kdo je oponentem závěrečné práce navrženým vedoucím katedry a proč garant nesouhlasil s navrženým oponentem dříve, než byl garantem navržen nový oponent.

Další důležitou otázkou je, kdy má být provedena auditní stopa pro uložení těchto informací žádosti k auditování. Ve fázi mohou zpracovatelé kdykoli upravovat instanci podle svého uvážení, dokud nepřevedou instanci do další fáze. Systém by měl umožnit uživateli upravovat instanci stejně jako upravovat dokument a ukládat aktuální obsah instance. Ale ukládání auditní stopy pro audit, kdykoli uživatel uloží instanci, nemá z hlediska administrativy velký význam. Zpracovatele zajímá pouze obsah instance před tím, než opustí předchozí fáze. Proto by náš systém měl ukládat auditní stopu pro audit pouze těsně před přechodem instance do další fáze.

Po uložení těchto informací musí být auditní stopy instance použity k rekonstrukci žádosti v libovolném okamžiku její historie. Z rekonstruované instance musí být patrné, kdo je editorem obsahu, v jakém čase a jaký byl obsah. Pro uložení těchto informací musíme navrhnout vhodný datový model, který bude popsán v následující části.

## Návrh datového modelu

Zobrazit celý relační datový model (zkráceně datový model) může být obtížné sledovat a pochopit, proto datový model rozdělíme na 3 části. V každé části se budeme podrobně zabývat.

Datový model byl normalizován ve třetí normální formě (3NF), jak ji zavedl Codd E. F. v [17] [18] a v této části bude pro stručnost představen pouze ve 3NF. Datový model jsme se zde také snažili co nejvíce zjednodušit, abychom se vyhnuli zbytečnému bloudění a nejednoznačnosti při řešení složitého a rozsáhlého problému. Datový model v reálné implementaci může být složitější, aby poskytoval více funkcí, ale měli jsme za to, že nemají vliv na jádro problému, který se snažíme v této práci řešit, proto je v následujících podkapitolách nebudeme popisovat.

K vizualizaci návrhu datového modelu jsme použili Entity Relationship Diagram (ERD) poskytovaný opensource softwarem StarUML. ERD ve StarUML lze použít k vizualizaci návrhu relačního datového modelu. Jeho nevýhodou je, že zobrazuje vztah pouze jako spojovací čáru mezi dvěma tabulkami bez uvedení, který sloupec jedné tabulky se odkazuje na druhý. Poskytuje však znaky, které lze použít k zobrazení odkazu mezi dvěma tabulkami, jsou to primární klíč (PK) a cizí klíč (FK). V ERD se používají i další znaky: Nullable (N), Unique (U), které označují odpovídající omezení v databázi. ERD pak lze použít jako blueprint pro vytvoření odpovídající kolekce tabulek v PostgreSQL pomocí SQLAlchemy. Dále budeme používat pouze termín "tabulka" místo "entita" (používaný v ERD) nebo "relace" (používaný v relačním datovém modelu), termín "sloupec" místo "atribut".

Kardinalita spojení dvou tabulek v ERD je znázorněna na Obrázek 13.

Text

Description automatically generated with low confidence

Obrázek 13 - ERD kardinalita

Zdroj: StarUML

Datové typy sloupců v ERD, jejich odpovídající datové typy v PostgreSQL a jejich hodnoty jsou uvedeny v následující tabulce.

Tabulka 1 - Datové typy datového modelu

| **Data type in ERD** | **Data type in PostgreSQL** | **Value** |
| --- | --- | --- |
| BIGINT | bigint | Integer from ‑9223372036854775808 to 9223372036854775807 |
| SMALLINT | smallint | -32768 to +32767 |
| VARCHAR(N) | character varying [(n)] | Variable-length (max. n) character string |
| TEXT | text | Variable unlimited length |
| BOOLEAN | boolean | True or False value |
| TIMESTAMP | timestamp with time zone | Both date and time with time zone |
| ENUM | enumerated type | A static set of values (will be listed in the following subsection) |

Zdroj: vlastní

Pro pojmenování jsme použili množné číslo podstatného jména s malým písmenem pro název tabulky, protože tabulka bude obsahovat mnoho záznamů. Pro pojmenování vztahů 1:1 a 1:N jsme použili slovesa pro podstatné jméno v jednotném čísle s malým písmenem, protože se používá k označení číselného vztahu mezi jedním záznamem v tabulce a jedním nebo více záznamy na druhé straně (vztah M:N se v relačním datovém modelu transformuje na dva vztahy 1:N pomocí mezilehlé tabulky, a proto se v následujících ERD nepoužívá). Malá písmena jsme použili také pro pojmenování atributů. Podtržítko (\_) se používá k oddělení slov.

### Users, groups, roles, positions

Diagram

Description automatically generated

Obrázek 14 - Datový model pro uživatelé, skupiny, role, pozice

Zdroj: vlastní

Jednou z nepostradatelných tabulek našeho systému jsou users. Dalšími nezbytnými tabulkami našeho systému jsou, jak je popsáno v oddílu 4.1, groups, roles, and positions. Každá entita má minimální počet atributů, např. id pro identifikaci každé instance entit, created\_at jako časové razítko, které udává, kdy byl záznam vytvořen, name pro jméno každého záznamu, ...

Tabulka users bude mít atribut user\_name a password pro uložení autentizačních informací každého uživatele.

Tabulka groups má vztah 1: N sama se sebou, aby modelovala hierarchickou strukturu skupiny v organizaci.

V tabulce roles je sloupec role výčtový typ, který obsahuje tři možné role, a to admin, handler a applicant. Tyto tři role jsou důležité pro autorizaci našeho systému. Uživatel s rolí admin může vytvářet, aktualizovat, mazat skupiny, role, pozice a přiřazovat pozice ostatním uživatelům. Role admin navíc jako jediná může definovat administrativní proces a odpovídající formulář. Role handler může buď vytvořit instanci formuláře a inicializovat proces, nebo může zpracovat instanci v tranzitní fázi. Role applicant může pouze vytvořit instance z formuláře a inicializovat proces. Sloupec name tabulky roles definuje role admin tak, aby odrážel skutečné role organizace, např. rektor, děkan, proděkan, vedoucí katedry, ...

Pozice, jak bylo uvedeno v kapitole 4.1, je odvozena od role a volitelně od skupiny. group\_id jako FK v tabulce positions může být nulové, jak je uvedeno ve výše uvedeném diagramu se znakem N, a pokud je group\_id nulové, pozice a role jsou ekvivalentní. To je užitečné, když chceme definovat obecnou pozici, jako je student, vedoucí apod. a ta nepatří do žádné skupiny. Obecně je účelem pozice, aby byla použita pro přidělení k zpracování fáze a určena pro přesměrování instance do další fáze. Dále v následujícím textu se používá role admin, handler nebo applicant pro zmiňování o pozicích, které .jsou odvozeny od těchto rolí.

Vztah mezi users a positions je M:N a je transformován do mezilehlé tabulky users\_positions.

### Administrativní proces a formulář

Proces správy je entitou našeho systému a může být tabulkou v datovém modelu. Administrativní proces bude obsahovat více fází a jak je omezeno v otevřené části této kapitoly, je doprovázen pouze jedním formulářem. To nás vede k otázce, zda můžeme k reprezentaci procesu správy použít formulář? V tomto smyslu neexistuje tabulka processes a tabulka forms budou obsahovat všechny fáze administrativního procesu (levý diagram na Obrázek 15). Tuto myšlenku jsme se pokusili realizovat a jedná se o jednodušší řešení, kdy je pro jeden správní proces definován pouze jeden formulář.

Pokud je pro jeden proces definováno více formulářů, můžeme přidat tabulkové procesy, jak je znázorněno na pravém Obrázku 15. Každý proces může mít více fází a více formulářů a každý formulář se používá v několika fázích procesu. Toto řešení lze považovat za sjednocení více podprocesů do jednoho procesu, kde podproces bude mít vlastní formulář s vlastními fázemi. Jedná se o rozšířené řešení výše uvedeného, protože je nutné přidat jednu tabulku se dvěma vztahy. To nám umožňuje realizovat jednoduché řešení při zachování možnosti rozšíření systému v budoucnu. V omezeném čase jsme implementovali pouze jednodušší řešení, a proto se v této části budeme zabývat pouze tímto jednoduchým řešením.

Diagram

Description automatically generated

Obrázek 15 - Jednodušší (vlevo) a rozšířené (vpravo) řešení datového modelu administrativního procesu

Zdroj: vlastní

Jak jsme uvedli v pododdíle [3.1.3](#_Choosing_database), sekvence fází může být uložena ve dvou tabulkách - phases a transitions. Vztah těchto dvou tabulek lze chápat tak, že tabulka phases má sama se sebou vztah many to many (Jedna fáze může mít mnoho dalších fází a jedna další fáze může mít mnoho fází), a proto se vztah transformuje na mezilehlou tabulku transitions se dvěma vztahy 1:N

Diagram

Description automatically generated

Obrázek 16 - Fáze před a po transformací

Zdroj: vlastní

Struktura formuláře je vytvořena ze sekcí a každá sekce obsahuje pole, jak je popsáno v pododdíle [4.1.3](#_Structure_of_a). Tabulky sections a fields musí být samostatné tabulky podle 1NF definované v [16] [17] a musí mít vztahy, jak je znázorněno v levém diagramu na Obrázek 17. V každé fázi se však budou zpracovávat některé sekce a každá fáze procesu patří do formuláře. Proto můžeme použít vztah 1:N mezi tabulkou phases a sections (pravý diagram na Obrázek 17) místo použití vztahu 1:N mezi forms and sections. Díky tomuto posunu vztahu můžeme vědět, ke kterému formuláři patří každá sekce, a zároveň přidáme vztah mezi phases a sections.

Diagram

Description automatically generated

Obrázek 17 - Přesun vztahu mezi formuláři a sekcemi na fáze a sekce

Zdroj: vlastní

Pozice jsou určeny pro fáze a jsou přiděleny pro sekce různých procesů pomocí admin a uživatel s určenou pozicí bude **potenciálním ředitelem** a **potenciálním zpracovatelem** těchto fází a sekcí. Kompletní datový model administrativního procesu je znázorněn na Obrázek 18.

Diagram

Description automatically generated

Obrázek 18 - Datový model administrativního procesu

Zdroj: vlastní

Sloupec phase\_type tabulky phases je výčtový typ, který zahrnuje 3 možné typy fází, a to begin, transit a end jak je uvedeno v pododdíle [4.1.1](#_What_is_a).

Sloupec order tabulek phases, sections a fields slouží k uspořádání jejich skutečného pořadí při prezentaci formuláře. Hodnota sloupce order je celé číslo od -32768 do +32767 (smallint v PostgreSQL) specifikované správcem, které rozhoduje o vzestupném pořadí fází, sekcí a polí v prezentaci formuláře a instance. Všechny sekce jsou uspořádány podle pořadí fází, které je obsahují, ve formuláři. Sekce jednotlivých fází jsou uspořádány podle pořadí jednotlivých sekcí. Pole každé sekce jsou uspořádána podle pořadí pole.

Sloupce public and obsolete tabulky forms slouží k označení, zda je formulář veřejný nebo zastaralý. Formulář je ve výchozím nastavení při vytvoření administrátorem neveřejný ( soukromý). Pouze soukromý formulář může admin číst, aktualizovat nebo mazat. Formulář, který je zveřejněn adminem, nelze upravovat. Uživatel může číst pouze veřejný a instancovaný formulář. Zastaralý formulář lze číst, ale nemůže být instancován jiným uživatelem. Formulář je označen jako zastaralý pouze adminem a pouze v případě, že je formulář aktuálně veřejný. S instancí zastaralého formuláře, který není aktuálně ve fázi ukončení, lze dále pracovat, dokud nedosáhne fáze ukončení.

### Instance vytvořeny od formuláře

Diagram

Description automatically generated

Obrázek 19 - Datový model instance

Zdroj: vlastní

Jak bylo uvedeno v pododdílu Výběr databáze, oddělujeme strukturu formuláře (sekce a pole) od údajů vyplněných uživatelem v jednotlivých polích formuláře, které nazýváme **obsahem**. Pro uložení obsahu musíme mít tabulku instances. Instance má však pro každé pole více obsahů, a proto je třeba obsah rozdělit do nové tabulky, contents, podle 1NF [16] [17] Vlastní data vyplněná uživatelem jsou uložena ve sloupce value tabulky contents. Sloupec updated\_at je časové razítko označující poslední aktualizaci value, což je rozhodující pro audit.

Administrativní proces začíná v okamžiku, kdy uživatel vytvoří instanci z formuláře a stane se ředitelem a zpracovatelem instance. Formulář může mít více instancí a instance může mít více ředitelů a zpracovatelů podle fází a sekcí definovaných adminem.

Instance se může v daném okamžiku objevit pouze v jedné fázi procesu. To platí v případě, že formulář je list papíru, který nemůže existovat ve dvou nebo více fázích. V určité fázi může zpracovatel(é) dané fáze upravit přiřazenou(é) sekci(e) a poté může ředitel přesměrovat instanci do další fáze. Než může ředitel přesměrovat instanci, musí určit zpracovatele a ředitele další fáze, jak je omezeno v pododdílu 4.1.4.

V tomto okamžiku je třeba zvážit jednu otázku: Kdy může ředitel instanci přesměrovat? Při implementaci API jsme se snažili zavést stavy instance při jejím průchodu fázemi, abychom tuto otázku vyřešili (sloupce current\_state tabulky instances a sloupce resolved tabulky contents na Obrázku 19). Pomocí těchto stavů můžeme stanovit striktní pravidlo, kdy může ředitel instanci přesměrovat. Například stav “full resolved” znamená, že všichni zpracovatelé jedné fáze již zpracovali svůj sekci (sekce) a označili svou sekci jako “resolved”. Ředitel může instanci přesměrovat do další fáze pouze tehdy, pokud je instance ve stavu “full resolved”. Například pokud byla instance odeslána do další fáze a zpracovatel další fáze našel v instanci nějakou chybu, musí všichni zpracovatelé této fáze označit instanci jako “resolved”, pak je ředitel schopen přesměrovat instanci zpět do fáze, kde k chybě došlo. Tento problém můžeme zmírnit zavedením více stavů, pokud je zavedeno více scénářů. Avšak místo toho, abychom se snažili zaměřit na každý scénář, použijeme velmi jednoduché a uvolněné pravidlo, které dokáže vyřešit každý scénář, tedy: kdykoli ředitel přesměruje instanci do další fáze, instance se nachází v další fázi bez ohledu na to, zda je instance “resolved” nebo ne. Pomocí tohoto pravidla také jednoduše provádíme auditní část našeho systému, která je popsána v následující podkapitole.

Obsah pole v instanci musí být vytvořen pouze jednou. Mohli jsme nechat instance\_id a field\_id stát PK tabulky contents a unikátní omezení PK zaručí jedinou existenci obsahu pole v instanci. Unikátní omezení však můžeme pro tato dvě pole vytvořit explicitně při vytváření tabulky nebo pozdější úpravě tabulky (Obrázek 20 ukazuje, jak upravit tabulku pro přidání unikátního omezení pro více sloupců). Pak je vytvořen sloupec id jako PK tabulky contents pro identifikaci obsahu pole v instanci. Tímto způsobem zaručíme jedinečné omezení obsahu pole v instanci a poskytneme jediný referenci tohoto obsahu pro jinou tabulku (jiná tabulka je obálka tabulky, kterou si představíme v další podkapitole).

A screenshot of a computer

Description automatically generated with medium confidence

Obrázek 20 - Přidání unikatního omezení pro tabulku contents explicitně v PostgreSQL

Zdroj: vlastní

Podobně má instance v libovolné fázi pouze jednoho ředitele a pro libovolnou sekci v instanci existuje pouze jeden manipulátor, a proto vytvoříme jedinečné omezení na instance\_id a phase\_id pro tabulku directors a další unikatní omezení na instance\_id a section\_id pro tabulku hadlers. Sloupec id tabulky directors a handlers slouží k jednoznačné identifikaci záznamů v těchto tabulkách.

Text

Description automatically generated

Obrázek 21 - Vytvoření tabulku directors a handlers s unikátním omezením na více sloupce v PostgreSQL

Zdroj: vlastní

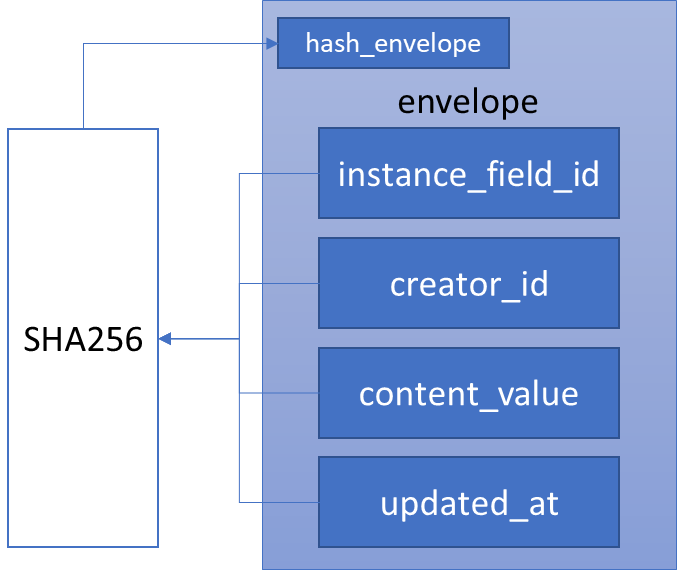
### Auditování a auditní stopa

Jak bylo uvedeno v oddílu 1.3, auditní stopa je chronologický záznam žádosti žadatele v každé fázi procesu administrace, který můžeme rekonstruovat a zkoumat posloupnost fází, kterými žádost prošla v předem definovaném procesu administrace.

Chronologický záznam žádosti žadatele znamená, že musíme ukládat údaje o žádosti v časovém pořadí. Abychom mohli rekonstruovat a zkoumat posloupnost fází, musíme uložit historii fází, kterými instance prošla. V každé fázi musíme načíst kopii instance a zkontrolovat celý její obsah, zpracovatele, kteří obsah naposledy upravili, a čas poslední úpravy. Teprve v okamžiku těsně před přechodem instance do další fáze se tyto údaje uloží a slouží jako auditní stopa. Akce vytvoření auditní stopy se nazývá **commitování** a auditní stopa se nazývá **commit** Instance je automaticky commitována poté, co ředitel tranzituje instanci do další fáze.

Jak bylo uvedeno v pododdílu 3.5.1 vidíme, že přístup systému Git k ukládání historie commitů je podobný našemu požadavku na historii instance v jednotlivých fázích. Chceme-li tento přístup použít v našem systému:

Nejprve musíme uvažovat, že každá hodnota obsahu v instanci je jako blob v systému Git, ale místo hashování pouze obsahu (a velikosti souboru) blobu jako v systému Git použijeme hashovací funkci, která vytvoří hash z id obsahu, id zpracovatele, který obsah naposledy upravil, hodnoty obsahu a časového razítka poslední úpravy zpracovatele před přechodem instance do další fáze. Hash a všechny jeho parametry vytvoří to, čemu říkáme **obálka**. Všimněte si, že místo ukládání id instance a id pole pro jednoznačnou identifikaci obsahu pole v instanci nám stačí odkazovat na obálku s jediným id obsahu, čímž snížíme počet sloupců tabulky. Id obsahu nám také pomáhá zabránit tomu, aby dvě stejné hodnoty obsahu dvou různých instancí modifikované ve stejnou dobu měly stejný hash. Sloupce creator\_id je zase id zpracovatele, který obsah naposledy upravil. Příjemce obsahu může být změněn ředitelem, jak je popsáno v pododdílu 4.1.4 a proto musí být uveden zpracovatel obsahu, aby bylo možné identifikovat, kdo obsah naposledy upravil. Časová značka je potřebná k určení, kdy byl obsah naposledy upraven, než je instance převedena do další fáze. Dvě stejné hodnoty obsahu editované stejným zpracovatelem, ale v různém čase, se stanou dvěma obálkami se dvěma různými hashi. Odpovídající tabulkou budou envelopes se sloupci hash\_envelope, content\_value a instance\_field\_id.

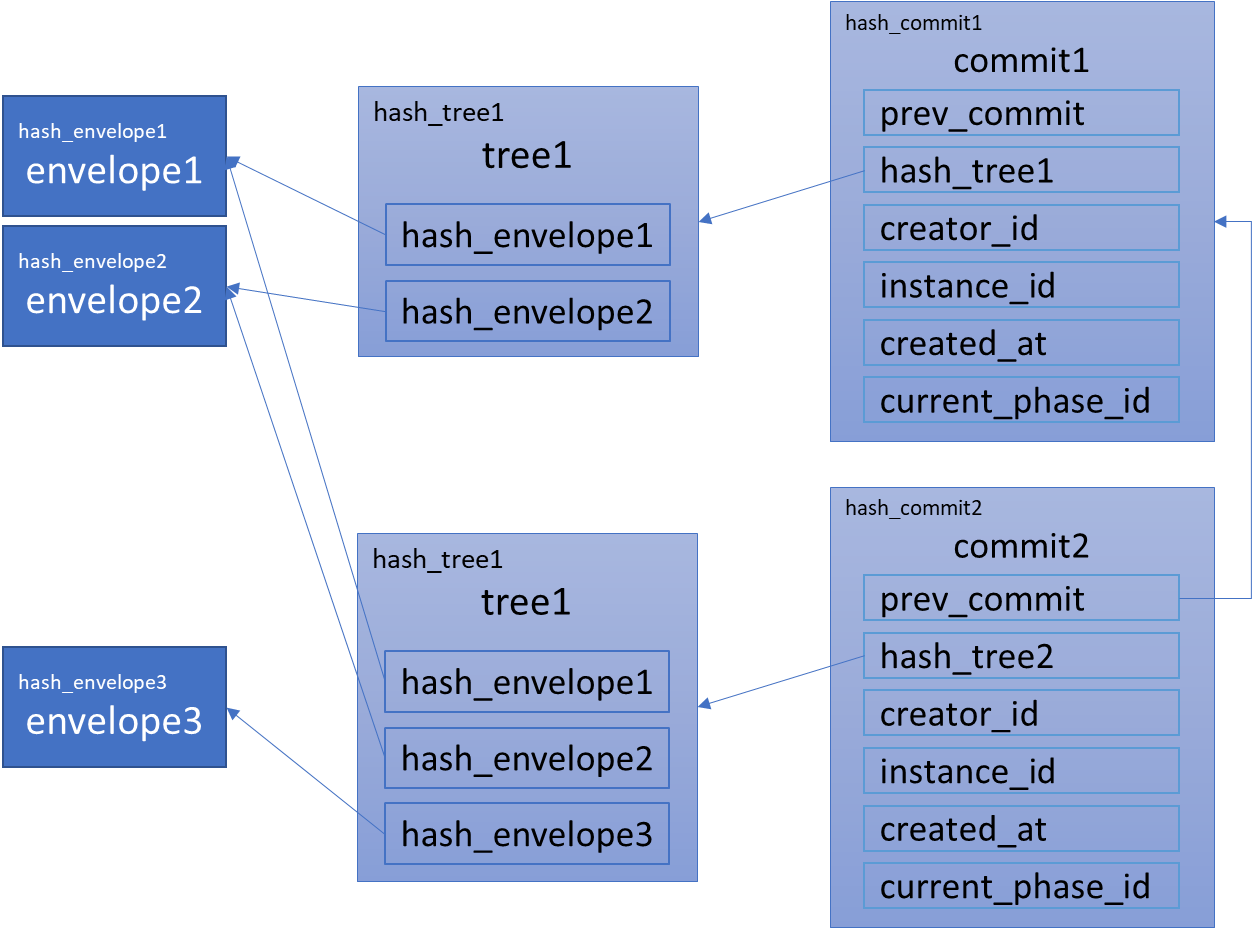


Obrázek 22 - Vytvoření envelope

Zdroj: vlastní

Všimněte si, že v tomto modelu nejsme pevně vázáni na použití konkrétního typu hašovací funkce. Pro pozdější implementaci rozhraní API jsme se však rozhodli použít jako hašovací funkci SHA256, protože je v době psaní této práce považována za bezpečnou proti kolizím [20]  [21] . Alternativou mohou být i jiné hashovací funkce, pokud jsou považovány za bezpečné.

Dále potřebujeme také strom jako Git, ale místo toho, abychom strom používali k ukládání názvu souborů a hashe blobu nebo názvu složek (adresářů) a hashe stromu, používáme strom pouze k ukládání hashe obálky. Strom je jako snímek instance v určité fázi procesu. Hash stromu je vytvořen ze všech hashů jeho obálek. Odpovídající tabulka bude mít názvy trees se sloupci hash\_tree.



Obrázek 23 - Příklad dvě commit

Zdroj: vlastní

Nakonec bude commit obsahovat hash předchozího commitu, pokud existuje, hash stromu, id tvůrce commitu (id ředitele), id instance, časové razítko vytvoření commitu, id fáze, ve které byl commit vytvořen. Tyto údaje se pak použijí k vytvoření hashe aktuálního commitu. Tabulka bude mít název commits se sloupci hash\_commit, prev\_commit, hash\_tree, creator\_id, instance\_id, created\_at, current\_phase\_id.

Kompletní datový model pro audit a auditní stopu je znázorněn na Obrázku 24 Kompletní datový model pro audit a auditní stopu je znázorněn na obrázku 23. Hodnota hash obálky, stromu a commitu je v databázi uložena ve formě 64 hexadecimálních číslic jako řetězec o délce 64 znaků.

Diagram

Description automatically generated

Obrázek 24 - Datový model pro auditování

Zdroj: vlastní

## Návrh API a implementace

### Vytvoření API pomocí FastAPI

Pro vytvoření jednoduchého rozhraní API v rozhraní FastAPI s jedním koncovým bodem:

* Musíme vytvořit instanci aplikace ze třídy FastAPI
* Vytvořit funkci, kterou FastAPI nazývá funkce operace cesty (Path Operation Function, POF)
* dekorovat POF pomocí dekorátoru operace

Kód v jazyce Python bude:

from fastapi import FastAPI

app = FastAPI()

@app.get("/forms/{form\_id}")

def get\_form(form\_id: int):

    return {"form\_id": form\_id}

FastAPI automaticky vytvoří soubor openapi.json, který je ve shodě se specifikací OpenAPI, a z tohoto souboru se vytvoří interaktivní dokument pomocí Swagger UI, jak je uvedeno na Obrázku 25:

Graphical user interface, application

Description automatically generated

Obrázek 25 - Interaktivní dokumentace vytvořen pomocí Swagger UI

Zdroj: vlastní

### Vzor zdroje, instance zdroje a souvisejícího zdroje

Při pokusu o vytvoření rozhraní API pomocí rozhraní FastAPI si uvědomíme vzor prostředku, instance prostředku a souvisejícího prostředku. Podle Fieldinga: “Any information that can be named can be a resource” [5] . Tuto myšlenku aplikujeme v našem systému a určíme, že zdroj, nebo jsme to konkrétně nazvali **instance zdroje**, bude odpovídat záznamu v každé tabulce. **Kolekce zdrojů** je kolekce instancí zdrojů. **Souvisejícím zdrojem** může být sloupec tabulky, záznam tabulky, kolekce záznamů nebo jakákoli informace odvozená z údajů jedné nebo mnoha tabulek, které souvisejí s instancí zdroje, např. uživatelské jméno instance uživatele, tvůrce formuláře, všechna pole formuláře, všichni účastníci instance, ...

Dále existují pouze 4 metody HTTP, které se v této práci používají ve spojení se třemi typy zdrojů (ostatní metody HTTP PUT, HEAD, CONNECT, OPTIONS, TRACE se nepoužívají), jsou to:

* Metoda GET se používá k získání vybrané kolekce zdrojů, instance zdroje nebo souvisejícího zdroje instance zdroje.
* Metoda POST se používá k vytvoření nové instance zdroje pro kolekci zdrojů z údajů obsažených v požadavku.
* Metoda PATCH se používá k aktualizaci atributu (atributů) instance zdroje, který odpovídá údajům uvedeným v požadavku.
* Metoda DELETE se používá k odstranění instance zdroje z kolekce zdrojů.

Výsledkem kombinace 3 typů prostředků a 4 metod HTTP je 6 typů obecných koncových bodů používaných v našem API, které jsou shrnuty v Tabulce 2. Spotřebitel API může zadat požadavek odpovídající každému koncovému bodu.

Tabulka 2 - 6 typů generického koncového bodů

|  |  |  |
| --- | --- | --- |
| HTTP metod | URL cesta | Název koncového bodu |
| GET | /{ rsc\_col } | Get Resource Collection |
| GET | /{ rsc\_col }/{ rsc\_id } | Get Resource Instance |
| GET | /{ rsc\_col }/{ rsc\_id }/{ rel\_rsc } | Get Related Resource |
| POST | /{ rsc\_col } | Post Resource Collection |
| PATCH | /{ rsc\_col }/{ rsc\_id } | Patch Resource Instance |
| DELETE | /{ rsc\_col }/{ rsc\_id } | Delete Resource Instance |

### Architektura API

V rozhraní API pro vytváření pomocí rozhraní FastAPI bude základní logika pro každý koncový bod implementována v těle každé funkce POF. Logika zahrnuje autentizaci uživatele, kontrolu omezení pro každý zdroj, ověření dat, zpracování dat s databází, přípravu odpovědi a její odeslání. Tyto logiky jsou implementovány prostřednictvím bázové třídy a jejích potomků, které jsem vytvořil nebo je poskytují jiné knihovny. Diagram tříd je znázorněn na Obrázku 26

SQLAlchemy poskytuje deklarativní bázovou třídu odvozenou od funkce declarative\_base(), kterou na Obrázku 26 označujeme jako BaseModel. BaseModel slouží ke zpracování dat s databází. Pro každou tabulku v naší databázi vytvoříme třídu zděděnou z této třídy BaseModel. Zděděná třída se nazývá Object Relational Mapper (ORM). Každá ORM je spojena s databázovou tabulkou a každá instance instancovaná z ORM je spojena se záznamem v této tabulce. ORM lze použít k vytvoření nebo odstranění tabulky jako DDL a používá se hlavně k vkládání, načítání, aktualizaci, mazání dat jako DML, ale na vyšší úrovni abstrakce. Pomocí ORM můžeme zpracovávat data v paradigmatu OOP v jazyce Python a bez použití jazyka SQL (i když použití ORM SQLAlchemy stále vyžaduje znalosti jazyka SQL).

Knihovna pydantic poskytuje třídu BaseModel, kterou na Obrázku 26 označujeme jako třídu BaseSchema[[2]](#footnote-2). Používá se k ověřování dat pomocí pythonovské typové anotace. Validace pomocí pydantic platí pro každý obecný případ použití, ale v našem systému používáme BaseSchema pouze dvěma způsoby: k validaci uživatelského vstupu v těle požadavku a k filtrování dat v odpovědi.

Diagram

Description automatically generated

Obrázek 26 - Diagram třídy pro vytvoření API

Zdroj: vlastní

Třída AuthorizationChecker slouží ke kontrole role uživatele při konkrétní operaci se zdrojem, například: pouze admin může vytvořit proces administrace a formulář, nebo ke kontrole, zda má uživatel právo vlastnit instanci zdroje, například: uživatel může změnit pouze své vlastní heslo. Třída AuthorizationChecker používá pro svou kontrolu oprávnění 4 schémata poskytovaná třídou BaseController, tj. post\_schema, patch\_schema, delete\_schema, response\_schema (na Obrázku 26 naznačeno jako asociace mezi těmito dvěma třídami).

BaseController je základní třída pro řízení požadavku, zpracování dat a přípravu odpovědi. Třída zděděná z BaseController se nazývá kontrolér (controller). Každý kontrolér bude mít svůj vlastní model, který odpovídá ORM. Řízení požadavku znamená, že 6 typů požadavků bude zpracováváno 6 odpovídajícími metodami kontroléru, které se používají pro generování POF pro FastAPI. 6 metod BaseControlleru implementují základní zpracování dat a připravuje odpověď bez specifického omezení systému. Kontrolér zděděný z BaseControlleru přepíše těchto 6 metod a implementuje vlastní zpracování dat a přípravu odpovědi. Zpracování dat znamená, že se musí zkontrolovat systémové omezení na zdroj, teprve potom se provede operace s daty (např. insert, select, update, delete) a commituje se do databáze. Systémové omezení je specifické pro každou problémovou doménu a v případě naší práce byla systémová omezení popsána v celé pododdílu 4.1 and 4.2. Například instance může být vytvořena pouze z veřejného formuláře nebo instance může být převedena do další fáze pouze v případě, že je specifikován ředitel. Příprava odpovědi znamená, že data vrácená z ORM musí být zkontrolována, zda se jedná o kolekci prostředků, instanci prostředku nebo primitivní datový typ, a před vrácením dat jako odpovědi musí být provedena filtrace dat z databáze pomocí schématu.

K validaci a filtrování se používají 4 typy schémat třídy BaseController a AuthorizationChecker. Schéma post\_schema a patch\_schema se používá pro ověřování vstupu uživatele v těle požadavků POST a PATCH a pro kontrolu role a/nebo vlastnictví uživatele při vkládání a aktualizaci instance zdroje. Schéma delete\_schema se používá pouze pro kontrolu role a/nebo vlastnictví uživatele při mazání instance prostředku v žádosti DELETE. Schéma response\_schema se používá pro filtrování dat z databáze před odesláním odpovědi uživateli.

Atribut related\_resources kontroléru je pole řetězců, ve kterém můžeme deklarovat, které jsou souvisejícími zdroji instance zdroje. Toto pole musí obsahovat pouze řetězec, který je buď veřejným atributem, veřejnou metodou, nebo veřejnou vlastností (property) odpovídající modelu v atributu model. Ve výchozím nastavení related\_resources obsahovat všechny veřejné atributy, veřejné metody a veřejné vlastnosti odpovídající třídy modelu.

Třídu Exception poskytuje Python. Z této třídy vytvoříme zděděné třídy pro vlastní výjimky. Každá výjimka je vyvolána při porušení autentizace, autorizace, ověření dat nebo omezení naší problémové domény. Naše vlastní výjimky jsou zachyceny v každé funkcí POF (na Obrázku 26 jsou označeny jako asociace mezi Exception a POFGenerator) a uživateli bude zaslána odpověď HTTP se stavovým kódem chyby.

BaseController je složen z BaseModel a BaseSchema a je agregován s Exception. POFGenerator je složen z BaseController. Má 6 metod, které generují POF pro FastAPI ze 6 metod každého kontroléru. Každá metoda generátoru je místem, kde spolu komunikují AuthenticationChecker a kontrolér a kde se zachycují vlastní výjimky. Kromě toho každá metoda generátoru POFGenerator přiřadí docstring odpovídající metody kontroléru, aby poskytla technickou dokumentaci každého koncového bodu API.

Výhodou tohoto návrhu je rychlé a snadné vytvoření a editace koncového bodu prostředku, oddělení zpracování dat s databází a systémových omezení, proto je snadné změnit kód v metodě kontroléru, když se změní systémová omezení.

Nevýhodou tohoto návrhu jsou schémata, která se musí vytvářet pro každou novou tabulku v databázi; koncový bod se musí explicitně definovat, pokud se neřídí konvencí 6 typů koncových bodů. Při implementaci rozhraní API jsme se však nejvíce setkali s tím, že změna systémového omezení a nevýhoda tohoto návrhu neměla při vytváření nebo změně koncového bodu velký vliv.

## Technická dokumentace

Dokumentace k rozhraní API musí každému uživateli rozhraní API poskytnout informace o tom, jak odeslat požadavek na rozhraní API a získat zpět odpověď a jak z rozhraní API vytvořit klientskou aplikaci (frontend nebo uživatelské rozhraní). Pro splnění tohoto úkolu rozdělíme dokumentaci na dvě části. V první části poskytujeme nízkoúrovňovou dokumentaci o tom, jak odeslat požadavek a získat zpět odpověď. Ve druhé části poskytujeme dokumentaci na vysoké úrovni o tom, jak kombinovat koncové body rozhraní API a vytvořit z tohoto rozhraní API klientskou aplikaci.

### Nizkoúrovňovou dokumentace

První část provedeme pomocí automaticky generovaného souboru openapi.json od FastAPI. Tento soubor lze použít v Swagger UI, což je interaktivní dokumentace pro odeslání požadavku a získání odpovědi. Každý spotřebitel rozhraní API, který získá kopii souboru openapi.json jej může použít k vygenerování interaktivní dokumentace s Swagger UI nebo jakékoli automaticky generované interaktivní dokumentace, která je ve shodě se standardem OpenAPI..

V nízkoúrovňové dokumentaci se zaměřujeme pouze na poskytování informací o: HTTP, parametr cesty, parametr dotazu, schéma požadavku, schéma odpovědi a popis koncového bodu API. Tyto informace postačují spotřebiteli k vytvoření požadavku a získání zpětné odpovědi na koncovém bodě. Příklad nízkoúrovňové dokumentace pro koncový bod API je uveden na Obrázku 27.

Graphical user interface, text, application

Description automatically generated

Obrázek 27 - Příklad nízkoúrovňové dokumentace pro koncového bodu

Zdroj: vlastní

Soubor openapi.json je poskytován v příloze této práce.

### Vysokoúrovňová dokumentace

Druhá část dokumentace však vyžaduje větší úsilí k dosažení. Musíme popsat všechny kroky, kterými se může spotřebitel API řídit a sestavit si vlastní klientskou aplikaci. Dokumentace na vysoké úrovni je uvedena v příloze této práce. V této dokumentaci se soustředíme pouze na důležité úkony, které splňují požadavky popsané v kapitole 1 této práce, těmi jsou:

* Jak definovat sekvenci fází (proces správy)
* Jak definovat, kdo je/bude zodpovědný za jednotlivé fáze procesu správy.
* Jak definovat strukturu formuláře.
* Jak vytvořit instanci z definovaného formuláře.
* Jak projít instanci procesem správy.
* Jak obnovit instanci v kterékoli fázi.

V další podkapitole je popsáno minimalistické uživatelské rozhraní jako příklad implementace klientské aplikace.

## Minimalistické uživatelské rozhraní

Abychom demonstrovali funkčnost API, použijeme React k vytvoření minimalistického uživatelského rozhraní s pomocí knihovny Cytoscape.js pro vizualizaci fáze a přechodu. Výsledek je znázorněn na obrázcích níže.

Obrázek 5 ukazuje, jak vytvořit fázi pro administrativní proces v uživatelském rozhraní. Vytvoření fáze zahrnuje zadání názvu, volitelně popisu fáze, typu fáze, určené pozice a pořadí fáze. Určená pozice je místem, kde definujeme odpovědnou osobu, která se stane ředitelem fáze. Typ fáze může být buď počáteční, tranzitní, nebo koncový, jak je uvedeno v oddílu 4.1.1. Pořadí fáze slouží k uspořádání pořadí sekcí, které do této fáze patří, ve formuláři a instanci.

Graphical user interface

Description automatically generated

Obrázek 28 - Vytvoření fáze v uživatelském rozhraní

Zdroj: vlastní

The Figure 27 ukazuje instanci vytvořenou z formuláře s názvem “APPLICATION FOR APPROVAL OF THE PROPOSAL TOPIC OF THE FINAL THESIS”. Všechny informace vyplňuje student v příslušné sekci a polích. Vedle názvu sekce vpravo je přiřazena pozice zpracovatele sekce. Černý rámeček označuje fázi formuláře. Informace o fázi a jejím přechodu jsou uvedeny v kontextovém menu vpravo nahoře u černého rámečku.

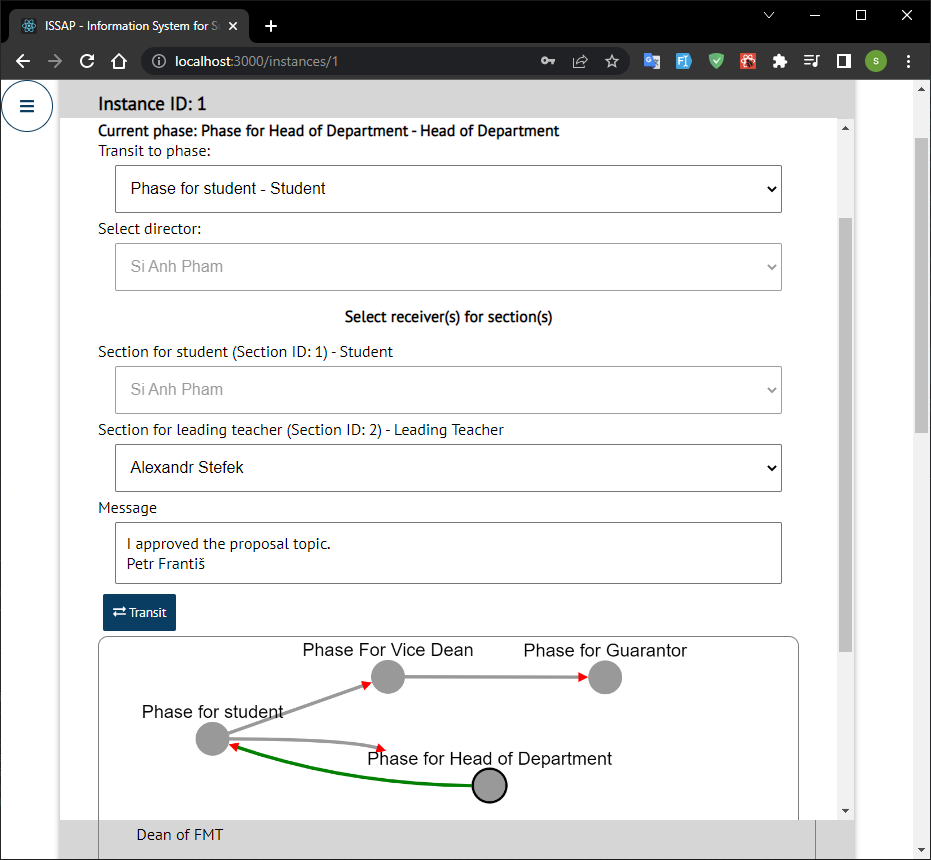
Graphical user interface, text, application

Description automatically generated

Obrázek 29 - Příklad instance v uživatelském rozhraní

Zdroj: vlastní

Další akce s instancí jsou k dispozici v kontextovém menu v pravém horním rohu instance, jedná se o přechod instance do další fáze, přijetí části instance, označení instance jako hotové, zobrazení historie revizí, čtení informací o fázích, přechodech, ředitelích a příjemcích instance, změnu příjemce instance. Na Obrázek 29 je zobrazeno uživatelské rozhraní při přechodu instance ředitelem ve fázi “Phase for Head of Department” do fáze “Phase for Student”, kde je již zadán ředitel další fáze a příjemce sekce pro studenta. Aktuální ředitel může zadat vedoucího práce pro odpovídající sekci další fáze. Ředitel může uvést zprávu pro commit, aby vyjádřil význam commitu nebo doplňující informace pro všechny účastníky instance.



Obrázek 30 - Přechod instance v uživatelském rozhraní

Zdroj: vlastní

Historie commitů instance je zobrazena na Obrázek 30. Každý commit obsahuje čas vytvoření commitu, fázi, ve které byl vytvořen, prvních 16 hexadecimálních číslic hashe, jméno tvůrce a zprávu.

Table

Description automatically generated

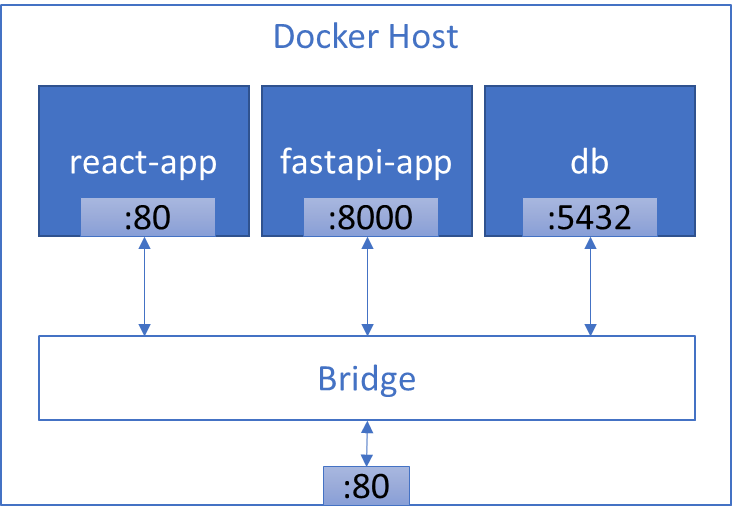
Obrázek 31 - Historie commitů (auditních stop) instance

Zdroj: vlastní

## Kontejnery s nástrojem Docker Compose

Když pomocí nástroje Docker Compose vytvoříme sadu kontejnerů, nástroj Compose ve výchozím nastavení nastaví bridge network, ve kterém se každý kontejner jako služba připojí k výchozí síti a je dosažitelný pro ostatní kontejnery v této síti a zjistitelný pro ně pod hostname shodným s názvem kontejneru. Síť vytvořenou nástrojem Compose si můžete představit na Obrázek 32. Máme 3 kontejnery, kterými jsou:

* db: je kontejner vytvořený z oficiálního obrazu postgres z PostgreSQL, který naslouchá na portu 5432.
* fastapi-app: kontejner rozhraní API, který je obsluhován službou Uvicorn a bude naslouchat na portu 8000.
* react-app: kontejner pro uživatelské rozhraní, který je obsluhován serverem Nginx na portu 80. Jedná se o jediný kontejner, který se mapuje na port 80 hostitele.



Obrázek 32 - Bridge network vytvořenp pomocí Compose

Zdroj: vlastní

Po spuštění příkazu npm run build, vytvoří React složku build, která obsahuje statické soubory, např. HTML soubor, JavaScript soubor, CSS soubor, ... Tuto složku může Nginx obsluhovat jako statickou webovou stránku.

Graphical user interface, text, application

Description automatically generated

Obrázek 33 - Adresář build vytvořen pomocí React

Zdroj: vlastní

Kromě toho lze Nginx použít jako reverzní proxy server pro předávání přenosů požadavků API aplikace react-app do aplikace fastapi-app. Minimální konfigurace pro Nginx jako webový server pro statické soubory a reverzní proxy bude následující:

http {

    server {

        listen 80;

        location /api/ {

            proxy\_pass http://fastapi-app:8000/;

        }

        include /etc/nginx/mime.types;

        location / {

            root /usr/share/nginx/html;

            try\_files $uri /index.html;

        }

    }

}

Server Nginx bude naslouchat na portu 80, jak bylo uvedeno výše. Provoz od API žádosti kontejneru react-app bude předán kontejnem fastapi-app na portu 8000. Obsah složky build bude umístěn do kořenového adresáře /usr/share/nginx/html a bude sloužit jako statický obsah. Při každém požadavku bude kontrolováno, zda zadaný soubor nebo adresář v URI existuje. Pokud neexistuje, provede Nginx interní přesměrování na soubor index.html v kořenovém adresáři. (Všimněte si, že musíme zahrnout typy MIME, aby mohl Nginx servírovat statické soubory, jako je CSS, jinak bude hlavička těchto souborů Content Type: text/plain; a CSS nebude funkční)

Komunikace mezi kontejnery a klientem je znázorněna na Obrázek 34. Klient provede požadavek HTTP na statický soubor index.html obsluhovaný systémem Nginx. Soubor index.html si vyžádá všechny soubory JavaScript a CSS v kořenové složce. Tyto 3 hlavní soubory vytvoří uživatelské rozhraní pro klienta. V závislosti na akci, kterou jsme v uživatelském rozhraní provedli, bude do Nginxu odeslán odpovídající požadavek API a Nginx předá požadavek kontejneru fastapi-app. Kontejner fastapi-app zpracuje požadavek, provede dotazy do db, a zkontroluje všechna omezení. Jakmile je odpověď připravena fastapi-app ji odešle do Nginxu a Nginx předá odpověď klientovi. Uživatelské rozhraní vykreslí data pro uživatele a uživatel uvidí změny akce, kterou provedl, v uživatelském rozhraní.

A screenshot of a computer

Description automatically generated with medium confidence

Obrázek 34 - Komunikace mezi klientem a kontejnery

Zdroj: vlastní

## Nezbytné výstupy

Zdrojový kód kontejneru poskytujícího službu autentizovaným uživatelům, dokumentace rozhraní API, jehož prostřednictvím je služba dostupná, a zdrojový kód kontejneru implementujícího uživatelské rozhraní jsou k dispozici na platformě GitHub na odkazu <https://github.com/Phamsianh/diplomova_prace>. Struktura repozitáře GitHub je rozdělena do 3 adresářů: backend, dokumentace, frontend podle pořadí požadovaného výstupu.

backend je adresář, do kterého vkládáme zdrojový kód služby (API) poskytující autentizovaným uživatelům. Soubor Dockerfile v tomto adresáři slouží k vytvoření obrazu pro spuštění kontejneru.

documentation obsahuje soubor openapi.json, který se používá jako nízkoúrovňová dokumentace, a soubor documentation.docx jako vysokoúrovňová dokumentace. V tomto adresáři lze nalézt i další dokumentaci a vizualizace.

Podobně jako adresář backend, adresář frontend obsahovat taký soubor Dockerfile který slouží k vytvoření obrazu pro spuštění kontejneru uživatelského rozhraní. Soubor Dockerfile je v tomto případě vícestupňové sestavení, ze kterého sestavíme naše uživatelské rozhraní React a Nginx jako webový server a reverzní proxy.

Nakonec se v kořeni úložiště nachází soubor docker-compose.yml, který se používá s nástrojem Docker Compose k vytvoření sady kontejnerů pro náš systém.

## Shrnutí

V prvním oddílu jsme se zabývali procesem správy, ve kterém jsme definovali následující omezení: Existuje pouze jedna počáteční fáze, jedna koncová fáze a více tranzitních fází; mezi dvěma fázemi mohou existovat pouze dva přechody a žádný přechod neexistuje mezi fází a sebou samou; každá fáze má pouze jednoho ředitele a případně více zpracovatelů; struktura formuláře obsahuje sekce, každá sekce obsahuje pole; každou sekci instance zpracovává vždy jeden zpracovatel; ředitel může určit nebo změnit zpracovatele sekce; ředitel aktuální fáze instance může instanci kdykoli tranzitovat a systém provede audit instance před jejím přechodem do další fáze.

Další oddíl popisuje datový model zahrnující 18 tabulek, jejich vztahy a omezení těchto tabulek. Tyto tabulky jsou users, groups, roles, positions, users\_positions, forms, sections, fields, phases, transitions, instances, instances\_fields, directors, receivers, envelopes, trees, trees\_envelopes, commits.

Třetí oddíl popisuje implementaci API pomocí FastAPI, která sleduje 6 typů obecných koncových bodů, kterými jsou Get Resource Collection, Get Resource Instance, Get Related Resource, Post Resource Collection, Patch Resource Instance and Delete Resource Instance. Pro vytvoření struktury API jsme definovali 3 třídy a použili je s třídami jiných knihoven, jsou to BaseController, AuthorizationChecker and POFGenerator, BaseModel od SQLAlchemy, BaseSchema od pydantic, APIRouter od FastAPI.

Technická dokumentace byla rozdělena na 2 části, nízkoúrovňovou dokumentaci, která popisuje, jak vytvořit požadavek a získat zpět odpověď koncových bodů API, a vysokoúrovňovou dokumentaci, která popisuje, jak kombinovat koncové body pro využití funkcionalit systému.

Ve čtvrtém oddílu bylo ukázáno uživatelské rozhraní vytvořené pomocí Reactu, které demonstruje funkcionality rozhraní API, jako je vytvoření formuláře, vytvoření instance, přechod instance a zobrazení historie commitů instance.

V šesté části byly popsány 3 kontejnery v podobě bridge network vytvořené nástrojem Docker Compose a komunikace mezi nimi. Tyto kontejnery jsou: db, fastapi-app, react-app.

Nakonec byla v oddílu 4.7 popsána struktura repozitáře GitHub, kam umisťujeme všechny potřebné výstupy této práce.

# Závěr

Cílem této práce je vytvořit informační systém (s přístupem přes web) s funkcionalitou podporující elektronizaci administrativy s prvky umožňující provést audit nad provedenými úkony. Tři výstupy jsou uvedeny na platformě GitHub na odkazu <https://github.com/Phamsianh/diplomova_prace>, jedná se o tyto výstupy:

* Zdrojový kód kontejneru poskytujícího službu autentizovaným uživatelům (adresář backend).
* Dokumentace k API, přes které je služba zpřístupněna (adresář documentation).
* Zdrojový kód kontejneru realizující uživatelské rozhraní (adresář frontend).

Přínosem této práce je vytvořený informační systém, který elektronizuje administrativní proces a umožňuje uživatelům: definovat sekvenci fází administrativního procesu; definovat odpovědného uživatele pro každou fázi; definovat strukturu formuláře složeného ze sekcí a polí; vytvořit instanci z definovaného formuláře a tranzitovat ji fázemi; vytvořit historii commitů instance a obnovit instanci v kterékoli fázi její historie.

V první kapitole jsme se seznámili s cílem této práce, požadavky na náš informační systém, zadáním pro dosažení cíle a potřebným výstupem práce. Dále je představen administrativní proces, audit a auditní stopa, aby čtenáři získali stručný přehled o problémové oblasti, která byla diskutována a řešena ve čtvrté kapitole.

Ve druhé kapitole jsme se seznámili s API obecně, s webovým aplikačním softwarem a webovým API zvláště, abychom dospěli k potvrzení toho, co je informační systém s přístupem přes web. Poté jsme zkoumali 3 typy webových API: RESTful API, RPC a GraphQL. Výsledkem je, že pro vytvoření našeho webového rozhraní API jsou použity standardy a technologie používané ve stylu RESTful API, tedy URL, HTTP, HTML, JS, CSS, JSON.

Třetí kapitola představila technologický zásobník, který se používá při vývoji našeho informačního systému, jsou to: PostgreSQL jako systém řízení relační databáze, FastAPI jako knihovna pro tvorbu webového API, React jako framework pro tvorbu uživatelského rozhraní, Docker jako technologie kontejnerizace, Git jako systém řízení verzí a GitHub jako platforma pro hostování zdrojového kódu. V této kapitole byla také představena inspirace systémem Git pro implementaci auditu.

Čtvrtá kapitola se podrobně zabývala administrativním procesem z hlediska požadavků na systém. V oddílech 4.2 až 4.6 byla také popsána implementace pokynů práce. Nezbytné výstupy jsou popsány v oddílem 4.7.

V této práci jsme se zcela soustředili na dosažení cíle práce s naplněním požadavků na informační systém, sledovali pokyny a poskytli potřebný výstup. Dostatečně lze konstatovat, že jsme splnili cíl práce, tedy s funkcionalitou podporující elektronizaci administrativy s prvky umožňující provést audit nad provedenými úkony.

V této práci jsme se zcela soustředili na dosažení cíle práce s naplněním požadavků na informační systém, sledovali pokyny a poskytli potřebný výstup. Dostatečně lze konstatovat, že jsme splnili cíl práce, tedy vytvořit informační systém (s přístupem přes web) s funkcionalitou podporující elektronizace administrativy s prvky umožňující provést audit nad provedenými úkony. Ovšem při implementaci systému jsme zjistili, že informační systém má ještě některá omezení ve funkcionalitách, které lze v budoucnu rozšířit, jako např. vytvoření správního procesu složeného z více dílčích procesů; sloučení procesů do jednoho procesu; proces mezi více organizacemi; automatické určení zpracovatelů pro sekce fáze v závislosti na skupině uživatele; upozornění pro uživatele, když instance čeká na příjem, zpracování nebo přechod; specifický typ dat pro každé pole, ... Právní účinek podpisů nebo pečetí zpracovatelů v elektronické podobě může být náročným problémem při implementaci. NAŘÍZENÍ EVROPSKÉHO PARLAMENTU A RADY (EU) č. 910/2014 [24], článek 25, bod 2 uvádí o právních účincích elektronických podpisů: “*Kvalifikovaný elektronický podpis má právní účinek rovnocenný vlastnoručnímu podpisu.*” a článek 35, bod 2 uvádí o právních účincích elektronické pečeti: “ *U kvalifikované elektronické pečeti platí domněnka integrity dat a správnosti původu těch dat, s nimiž je kvalifikovaná elektronická pečeť spojena.*” Aby instance formuláře měla v budoucnu právní účinek, měl by informační systém směřovat k integraci s kvalifikovaným poskytovatelem služeb vytvářejících důvěru, který uživatelům v systému poskytuje kvalifikované certifikáty pro vytváření kvalifikovaného elektronického podpisu nebo kvalifikované elektronické pečeti.

# Seznam použité literatury

1. HAYES, Rick Stephan. *Principles of auditing: an introduction to international standards on auditing*. 2nd ed. Harlow: Financial Times Prentice Hall, 2005. ISBN 0273684108.
2. *Guidelines for auditing management systems*. 3rd ed. Switzerland: International Organization for Standardization, 2018.
3. *Security and Privacy Controls for Information Systems and Organizations* [online]. National Institute of Standards and Technology. 2020. Dostupné z: doi:10.6028/nist.sp.800-53r5
4. CNSS 4009. *Committee on National Security Systems (CNSS) Glossary* [online]. Committee on National Security Systems. 2022. Dostupné z: <https://www.cnss.gov/CNSS/issuances/Instructions.cfm>
5. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures* [online]*.* 2000. Dostupné z: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
6. BLOCH, Joshua. *A Brief, Opinionated History of the API* [online]*.* 2018. Dostupné z: <https://www.infoq.com/presentations/history-api/>
7. STAIR, Ralph M. a George W. REYNOLDS. *Fundamentals of Information Systems*. 6th ed. Boston, USA: Course Technology, Cengage Learning, 2012. ISBN 0840062184.
8. BERNERS-LEE, T., R. FIELDING a L. MASINTER. *Uniform Resource Identifier (URI): Generic Syntax* [online]. B.m.: RFC Editor. 2005. Dostupné z: doi:10.17487/rfc3986
9. FIELDING, R., J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH a T. BERNERS-LEE. *Hypertext Transfer Protocol -- HTTP/1.1* [online]. B.m.: RFC Editor. 1999. Dostupné z: doi:10.17487/rfc2616
10. Facebook, Inc. *GraphQL* [online]*.* 2021. Dostupné z: <https://spec.graphql.org/October2021/>
11. OpenAPI Initiative. *OpenAPI specification v3.1.0* [online]. 2021. Dostupné z: <https://spec.openapis.org/oas/v3.1.0.html>
12. MogoDB, *Understanding the Different Types of NoSQL Databases* [online]*.* Dostupné z: <https://www.mongodb.com/scale/types-of-nosql-databases>
13. Docker. *Use containers to Build, Share and Run your applications* [online]*.* Dostupné z: <https://www.docker.com/resources/what-container>
14. POULTON, Nigel. *Docker Deep Dive*, *Zero to Docker in a single book.* 2020th edition*.* ISBN 1521822808.
15. React. *Reconciliation* [online]*.* Dostupné z: <https://reactjs.org/docs/reconciliation.html>
16. CODD, E. F. *A Relational Model of Data for Large Shared Data Banks,* IBM Research Laboratory, San Jose, California, 1970.
17. CODD E. F. *Further Normalization of the Data Base Relational Model,* IBM Research Laboratory San Jose, California, 1971.
18. Univerzita Obrany v Brně. *STUDIJNÍ ROZKAZ DĚKANA FAKULTY VOJENSKÝCH TECHNOLOGIÍ*, Zadávání a zpracování závěrečných prací v AR 2021/22. Brno, 2020.
19. CHACON S. a B. STRAUB. Pro Git, 2021, <https://git-scm.com/book/en/v2>
20. BLUECRYPT. *Cryptographic key length recommendation* [online]. Dostupné z: <https://www.keylength.com/>
21. NÁRODNÍ ÚŘAD PRO KYBERNETICKOU A INFORMAČNÍ BEZPEČNOST. *Minimální požadavky na kryptografické algoritmy: doporučení v oblasti kryptografických prostředků* [online]. Verze 1.0. Brno, 2018. Dostupné z: <https://nukib.cz/download/uredni-deska/Kryptograficke_prostredky_doporuceni_v1.0.pdf>
22. FIELDING, Roy Thomas. REST APIs must be hypertext-driven [online]. 2008. Dostupné z: <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>
23. RICHARDSON L., M. AMUNDSEN. *RESTful Web APIs*. Sebastopol, California: O’Reilly Media, Inc. 2013, ISBN: 9781449358068.
24. *NAŘÍZENÍ EVROPSKÉHO PARLAMENTU A RADY (EU) č. 910/2014 ze dne 23. července 2014 o elektronické identifikaci a službách vytvářejících důvěru pro elektronické transakce na vnitřním trhu a o zrušení směrnice 1999/93/ES*. EVROPSKÝ PARLAMENT A RADA EVROPSKÉ UNIE, 2014. <https://eur-lex.europa.eu/legal-content/CS/TXT/?qid=1588426886500&uri=CELEX:32014R0910>
25. JIN, Brenda, Saurabh SAHNI a Amir SHEVAT. *Designing Web APIs.* Sebastopol, CA: O’Reilly Media, Inc. 2018. ISBN: 9781492026921.
26. Neo4j. *The Neo4j Cypher Manual v4.4*, Introduction [online]. Dostupné z: <https://neo4j.com/docs/cypher-manual/current/introduction/>

# Přílohy

## Vysokoúrovňová dokumentace k API

Vysokoúrovňová dokumentace uvádí jak kombinovat koncové body pro vytvoření klientská aplikaci. Klientská aplikace musí poskytovat následující funkcionality:

* Definovat administrativní proces, což je sekvence fází.
* Definovat odpovědné uživatele pro každou fázi procesu.
* Definovat strukturu formuláře, což jsou sekce a pole.
* Autentizovaný uživatel může z formuláře vytvořit instanci.
* Instance může procházet fázemi.
* Rekonstruovat instanci v libovolné fázi v historii commitů.

Tyto funkcionality jsou rozebrány v odpovídajících oddílech níže.

Pro označení koncového bodu používáme metody HTTP a cestu v URL, např: POST /forms. Dokumentace o tom, jak vytvořit požadavek a získat zpět odpověď, je uvedena v nízkoúrovňové dokumentaci v souboru openapi.json. V této vysokoúrovňové dokumentaci popisujeme pouze to, jak kombinovat koncové body, abychom získali správné funkcionality navrženého systému.

### Definovat sekvenci fází (administrativní proces)

Kroky pro vytvoření administrativního procesu:

1. Vytvoření formuláře
2. Vytvoření fází
3. Vytvoření přechodů

Pro vytvoření sekvence fází musíme nejprve vytvořit formulář. K vytvoření formuláře použijeme koncový bod POST /forms. Následuje příklad těla odpovědi ve formátu JSON:

**{**

**"id": 1,**

**"created\_at": "2022-03-21T14:46:04.566020+01:00",**

**"name": "Proposal topic for thesis",**

**"creator\_id": 1,**

**"public": false,**

**"obsolete": false**

**}**

V odpovědi máme id nového formuláře, ze kterého můžeme vytvořit fáze pro tento formulář.

Pro vytvoření fáze pro formulář použijeme koncový bod POST /phases, ve kterém do pole form\_id těla požadavku zadáme id formuláře. Každý požadavek vrátí informace o nové fázi včetně id fáze, které použijeme k vytvoření přechodů procesu. Příklad těla odpovědi je následující:

**{**

**"id": 1,**

**"created\_at": "2022-03-21T14:06:37.162Z",**

**"form\_id": 1,**

**"name": "Phase for student",**

**"description": "string",**

**"position\_id": 11,**

**"phase\_type": "begin",**

**"order": 1**

**}**

*Poznámka: vytvoření fáze zahrnuje definici odpovědného uživatele pro tuto fázi. Viz následující pododdíl.*

K vytvoření přechodu použijeme koncový bod POST /transitions. V něm zadáme id zdrojové fáze do pole from\_phase\_id, a id cílové fáze do pole to\_phase\_id těla požadavku.

Z nejlepší praxe vyplývá, že bychom měli všechny fáze a jejich přechody naplánovat v administrativním procesu a před jeho vytvořením.

Všechny fáze a přechody formuláře můžeme zobrazit prostřednictvím koncových bodů GET /forms/{form\_id}/phases and GET /forms/{form\_id}/transitions.

*Poznámka: Administrativní proces může vytvořit pouze admin.*

### Definovat odpovědné uživatele pro každou fázi administrativního procesu

Víme, že pro každou fázi existuje pouze jeden ředitel a pro každý sekci této fáze pouze jeden zpracovatel. Odpovědnými uživateli jsou ředitel a zpracovatelé. Pro definování ředitele a zpracovatelů pro danou fázi používáme pozici, která označuje, že ředitelem nebo zpracovatelem fáze se může stát pouze uživatel, který držel odpovídající pozici. Říkáme: pozice je určena pro fázi a pozice je přidělena pro sekci.

Nejprve musíme získat všechny pozice v systémech pomocí koncového bodu GET /positions. Příklad těla odpovědi z tohoto koncového bodu je následující:

**[**

**...**

**{**

**"id": 11,**

**"created\_at": "2022-03-20T12:27:58.917751+01:00",**

**"name": "Student of 2-CIT-C",**

**"group\_id": 7,**

**"role\_id": 8,**

**"creator\_id": 1**

**},**

**...**

**]**

Id pozice se používá k zadání do pole positions\_id v těle požadavku koncového bodu POST /phases k určení pozice pro tuto novou fázi. Následuje příklad těla požadavku na vytvoření fáze:

**{**

**"form\_id": 1,**

**"name": "Phase for student",**

**"description": null,**

**"position\_id": 11,**

**"phase\_type": "begin",**

**"order": 1**

}

Podobně zadáme id pozice do pole position\_id v těle požadavku koncového bodu POST /sections abychom přidělili pozici pro tuto novou sekci. Příklad těla požadavku pro vytvoření sekce je následující:

**{**

**"name": "Section for student of 2-CIT-C",**

**"phase\_id": 1,**

**"position\_id": 11,**

**"order": 1**

**}**

*Poznámka: pouze admin může určit a přiřadit pozice pro fáze a sekce procesu administrace.*

### Definovat strukturu formuláře

Struktura formuláře zahrnuje sekce a pole. Každá fáze formuláře obsahuje sekce a každá sekce obsahuje pole.

Chceme-li vytvořit sekci, musíme nejprve vědět, do které fáze nová sekce patří. Id fáze pak použijeme jako hodnotu pole phase\_id v těle požadavku koncového bodu POST /sections. Příklad těla odpovědi z tohoto koncového bodu je následující:

**{**

**"id": 1,**

**"name": "Section for student of 2-CIT-C",**

**"created\_at": "2022-03-20T12:28:00.367400+01:00",**

**"phase\_id": 1,**

**"position\_id": 11,**

**"order": 1**

**}**

*Poznámka: vytvoření sekce zahrnuje definici odpovědného uživatele pro tuto sekci fáze. Viz předchozí pododdíl.*

Pro vytvoření pole použijeme koncový bod POST /fields, ve kterém jako hodnotu pole section\_id uvedeme id sekce, čímž označíme, že nové pole patří do této sekce. Příklad odpovědi z tohoto koncového bodu je následující:

**{**

**"id": 1,**

**"created\_at": "2022-03-20T12:28:00.367400+01:00",**

**"name": "Field for student of 2-CIT-C",**

**"section\_id": 1,**

**"order": 1**

**}**

*Poznámka: Strukturu formuláře může definovat pouze admin..*

### Vytvořit instanci z definovaného formuláře

Pro vytvoření instance z formuláře použijeme koncový bod POST /instances, ve kterém zadáme id formuláře jako hodnotu pole form\_id v těle požadavku. Pouze uživatel s pozicí určenou pro počáteční fázi formuláře může vytvořit instanci. Příklad těla požadavku tohoto koncového bodu je následující:

**{**

**"form\_id": 1**

**}**

Příklad těla odpovědi je následující:

**{**

**"id": 1,**

**"created\_at": "2022-03-20T12:28:01.652561+01:00",**

**"updated\_at": null,**

**"form\_id": 1,**

**"current\_phase\_id": 1,**

**"creator\_id": 1,**

**"current\_state": "initialized"**

**}**

Po vytvoření instance se její tvůrce stává ředitelem počáteční fáze a zpracovatelem všech počátečních sekcí, které jsou přiděleny pro pozici tvůrce. Veškerý obsah těchto sekcí se automaticky vytvoří s hodnotou prázdného řetězce. To je nezbytné pro vytvoření hashe obálky.

Pro získání všech ředitelů a příjemců instance použijeme koncový bod GET /instances/{instance\_id}/directors a GET /instances/{instance\_id}/receivers.

Pro získání veškerého obsahu instance použijeme koncový bod GET /instances/{instance\_id}/instances\_fields.

*Poznámka: Pouze účastníci instance mohou získat související zdroj prostředku instance.*

### Přechod instance administrativním procesem

Pro přechod (přesměrování) instance do další fáze použijeme koncový bod PATCH /instances/{instance\_id}. V těle požadavku tohoto koncového bodu musíme specifikovat:

* Id další fáze jako hodnotu pole current\_phase\_id.
* Id uživatele, který se stane ředitelem další fáze, jako hodnotu pole director\_id.
* Id sekcí a id uživatelů, kteří se stanou zpracovateli sekcí následující fáze, jako dvojice klíč/hodnota v poli receivers.
* Zpráva pro commit, která se automaticky vytvoří při přechodu instance do další fáze, jako hodnota v poli message.

Příklad těla požadavku je následující:

**{**

**"transit": {**

**"current\_phase\_id": 2,**

**"receivers": {**

**"3": 5**

**},**

**"director\_id": 5,**

**"message": "The proposal topic for final thesis of Pham Si Anh"**

**}**

**}**

*Poznámka: Pouze ředitel aktuální fáze může tranzitovat (přesměrovat) instanci do další fáze. Ředitel musí být specifikován, ale nemusí být specifikovány všechny obsluhy sekce.*

Uživatel s určenými pozicemi se nazývá potenciální ředitel fáze. Uživatel s pozicí určenou pro sekci se nazývá potenciální příjemce (nebo zpracovatel) sekce. Všechny potenciální ředitele fáze můžeme získat pomocí koncového bodu GET phases/{phase\_id}/potential\_directors a všechny potenciální příjemce sekce můžeme získat pomocí koncového bodu GET /sections/{section\_id}/potential\_receivers nebo GET /sections/{section\_id}/potential\_handlers.

Příjemce může zpracovávat instance pomocí stejného koncového bodu jako přechod instance, PATCH /instances/{instance\_id}. Tělo požadavku se však liší. Příjemce pouze označí pole handle to true. Příklad je uveden níže:

**{**

**"handle": true**

**}**

Pokud byla instance zpracována příjemcem, stane se příjemce zpracovatelem sekcí a vytvoří se veškerý obsah těchto sekcí přiřazených tomuto příjemci. Tento obsah může upravovat pouze jeho zpracovatel.

### Obnovit instanci v jakékoli fázi

Commit je automaticky vytvořena systémem, když ředitel přesměruje instanci do další fáze. Účastníci instance mohou získat všechny revize instance pomocí koncového bodu GET /instances/{instance-id}/commits. Příklad historie commitů je uveden níže, kde obsahuje pouze 1 commit.

**[**

**{**

**"hash\_commit": "f92892e32870eac834ba245a553997440226dbc9b41407a3cda7ba94c065607e",**

**"prev\_hash\_commit": null,**

**"hash\_tree": "fd5d2c21ed200ca30d8d92ec5afffe465888741a3da64d740fc9c9b535c7bfdb",**

**"creator\_id": 1,**

**"instance\_id": 1,**

**"created\_at": "2022-04-06T12:07:01.354591+02:00",**

**"current\_phase\_id": 1,**

**"message": "The proposal topic for final thesis of Pham Si Anh"**

**}**

**]**

Abychom mohli rekonstruovat instanci v libovolné fázi historie commitů, musíme nejprve načíst veškerý aktuální obsah instance a poté musíme načíst všechny obálky commitu instance. Klientská aplikace může použít hodnotu obsahu těchto obálek se strukturou formuláře k injektování do obsahu příslušných polí. Pro získání veškerého aktuálního obsahu instance použijeme koncový bod GET /instances/{instance\_id}/instances\_fields a pro získání všech obálek commitu instance použijeme koncový bod GET /commits/{hash\_commit}/envelopes. Příklad těla odpovědi pro získání veškerého obsahu instancí je uveden níže:

**[**

**{**

**"id": 1,**

**"created\_at": "2022-04-06T12:00:53.909711+02:00",**

**"updated\_at": "2022-04-06T12:01:50.279872+02:00",**

**"instance\_id": 1,**

**"field\_id": 1,**

**"creator\_id": 1,**

**"value": "Bc. Pham Si Anh",**

**"resolved": true**

**},**

**...**

**]**

Příklad těla odpovědi pro načtení všech obálek revize je uveden níže..

**[**

**{**

**"hash\_envelope": "eea27c6235e8866d1184cf7f3d18dae3efc3dd24e33b80db2f6bb0f4bbf780f8",**

**"instance\_field\_id": "1",**

**"creator\_id": 1,**

**"content\_value": "Bc. Pham Si Anh",**

**"updated\_at": "2022-04-06T12:01:50.279872+02:00"**

**},**

**...**

**]**

1. Jiné media type uvádí v <https://www.iana.org/assignments/media-types/media-types.xhtml> [↑](#footnote-ref-1)
2. Pro přehlednost BaseModel pro databázi, BaseSchema pro validaci [↑](#footnote-ref-2)