**UNIVERSITY OF DEFENSE**

**FACULTY OF MILITARY TECHNOLOGY**

**Study Program: Defense and Security Technology**

Study field: Communication and Information Technology

Ev. number:

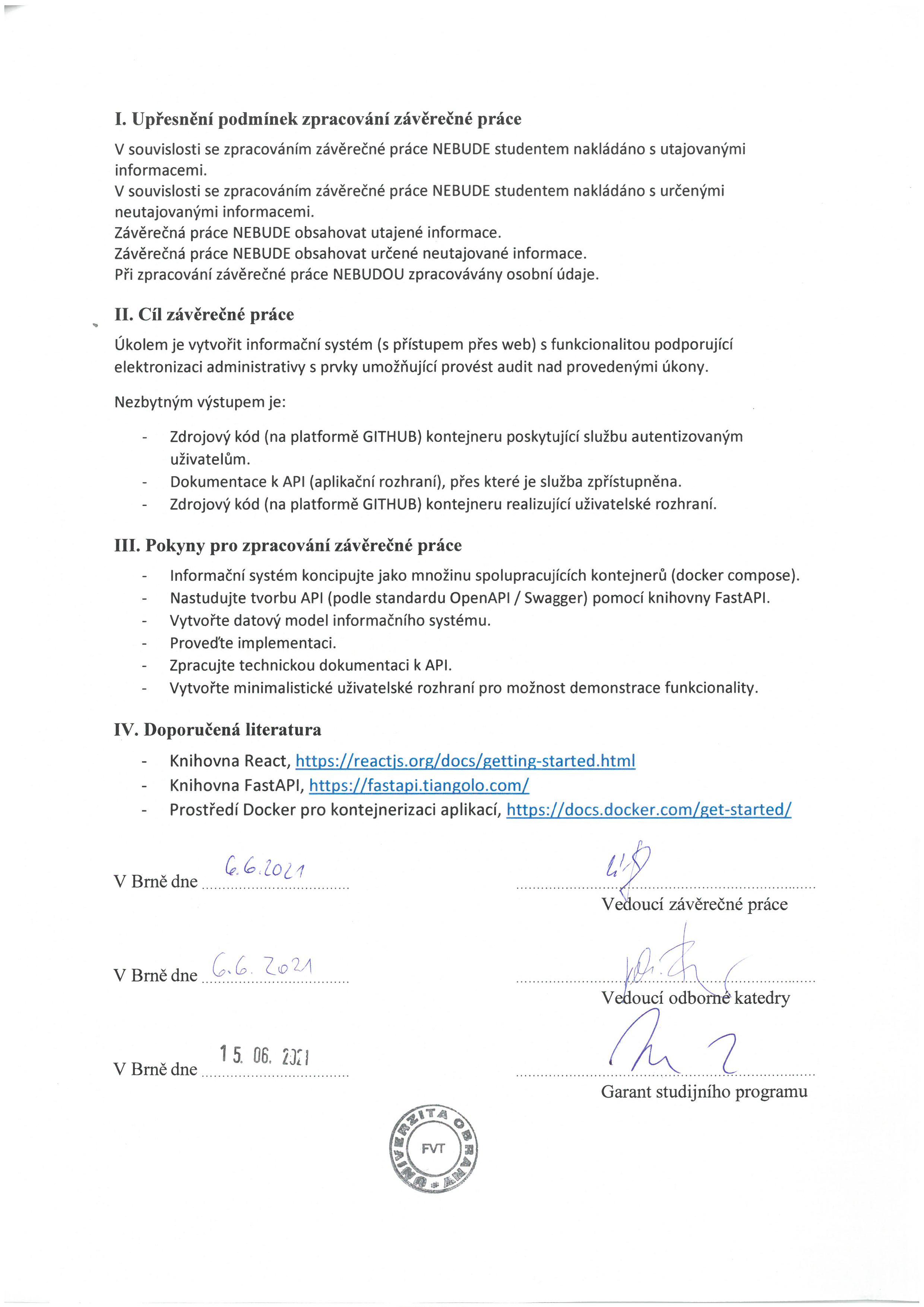
**MASTER’S THESIS**

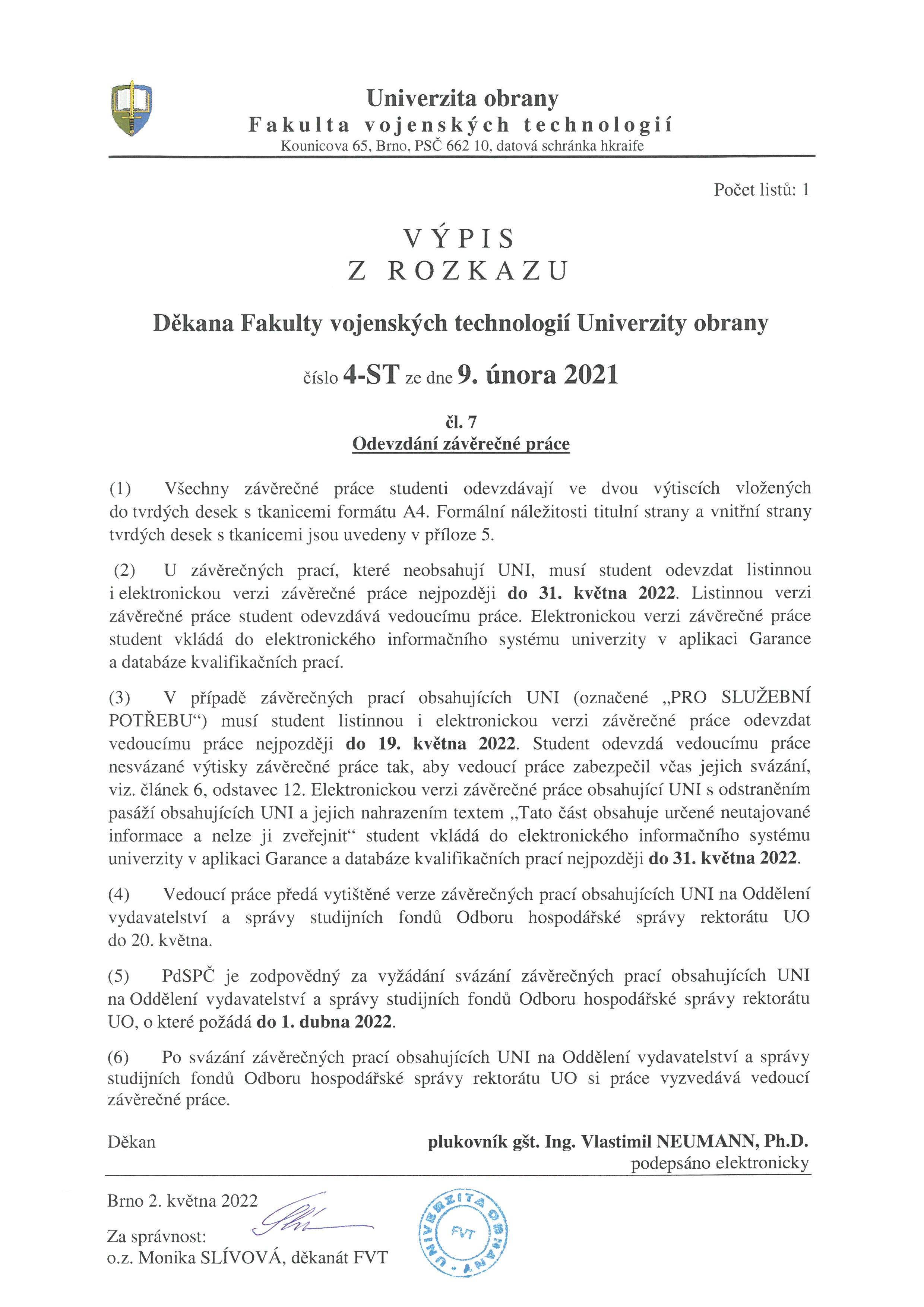
**INFORMATION SYSTEM WITH AUDIT TRAIL FOR SUPPORTING ADMINISTRATION PROCESS MANAGEMENT**

Author: Bc. Pham Si Anh

Supervised by: Prof. Dr. Eng. Alexandr Štefek

BRNO 2022





Acknowledgement

Foremost, I would like to express my deep and sincere gratitude to my supervisor, Professor Alexandr Štefek, for continuous support, his patience, enthusiasm, immense knowledge. I am extremely grateful for his time, even in weekend, spent to listen to my idea, to feedback, suggest, and orient me to achieve the goal; for his empathy with the work that I got from school; and for his always elegant, gentle, and clear explanation from a ton of questions of mine. It was an honor to work and study under his guidance.

I would like to give my thankfulness to my parents for their love, caring, and sacrifices for educating and preparing me for my future.

I would also like to thank to The Vietnamese People’s Army, Le Quy Don Technical University and University of Defense in Brno for the opportunity, conditions, and facilities to study, research and improve my knowledge.

Last but not least, to my best friend, my lover and my soulmate, Bui Bao Tran, who always gives me the motivation in the most stressful time, who always shows me the sympathy and compassion in her heart, who always gives me interesting ideas when I am in a deadlocked thought and who gives me a goal to try my best in study and life, thank you for always being with me.

Affidavit

I declare that I have prepared my master’s thesis independently under the guidance of the thesis supervisor and using professional literature and other information sources, which are listed in the bibliography.

Brno 27.05.2022

|  |
| --- |
| ……………………………………..  Name, Surname |

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, Application Programming Interface (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.

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, Application Programming Interface (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

Contents

[Abbreviations 11](#_Toc104279594)

[Figures 13](#_Toc104279595)

[Tables 15](#_Toc104279596)

[1 Introduction 16](#_Toc104279597)

[1.1 Goal and assignments for this thesis 16](#_Toc104279598)

[1.2 What is administration process? 17](#_Toc104279599)

[1.3 What is audit and audit trail? 18](#_Toc104279600)

[1.4 Thesis’s structure 19](#_Toc104279601)

[2 Web API 21](#_Toc104279602)

[2.1 Application Programming Interface (API) 21](#_Toc104279603)

[2.2 Web-based application software and Web API 22](#_Toc104279604)

[2.3 Information system with access via web 22](#_Toc104279605)

[2.4 REST and RESTful API 23](#_Toc104279606)

[2.4.1 REST 23](#_Toc104279607)

[2.4.2 RESTful API 23](#_Toc104279608)

[2.5 RPC 25](#_Toc104279609)

[2.6 GraphQL 26](#_Toc104279610)

[2.7 Other type of web API 27](#_Toc104279611)

[2.8 Summary 28](#_Toc104279612)

[3 Technology stack 29](#_Toc104279613)

[3.1 Types of databases 29](#_Toc104279614)

[3.1.1 Relational database 29](#_Toc104279615)

[3.1.2 Non-relational database 30](#_Toc104279616)

[3.1.3 Choosing database 32](#_Toc104279617)

[3.2 FastAPI 34](#_Toc104279618)

[3.2.1 FastAPI 34](#_Toc104279619)

[3.2.2 SQLAlchemy 35](#_Toc104279620)

[3.2.3 Swagger specification and OpenAPI specification 35](#_Toc104279621)

[3.3 React 36](#_Toc104279622)

[3.4 Docker 37](#_Toc104279623)

[3.4.1 Docker and container 37](#_Toc104279624)

[3.4.2 Networking in Docker and Docker Compose 38](#_Toc104279625)

[3.5 Git and GitHub 40](#_Toc104279626)

[3.5.1 The inspiration from Git 40](#_Toc104279627)

[3.6 Summary 42](#_Toc104279628)

[4 Discussion and implementation 43](#_Toc104279629)

[4.1 Administration process 43](#_Toc104279630)

[4.1.1 Phases in administration process 44](#_Toc104279631)

[4.1.2 The designated position and director 48](#_Toc104279632)

[4.1.3 Structure of a form 49](#_Toc104279633)

[4.1.4 Instance and its transition 50](#_Toc104279634)

[4.1.5 Audit and audit trails 51](#_Toc104279635)

[4.2 Data model design 52](#_Toc104279636)

[4.2.1 Users, groups, roles, positions 55](#_Toc104279637)

[4.2.2 Administration process and form 56](#_Toc104279638)

[4.2.3 Instances of a form 60](#_Toc104279639)

[4.2.4 Audit and audit trail 62](#_Toc104279640)

[4.2.5 Query examples 66](#_Toc104279641)

[4.3 API design and implementation 73](#_Toc104279642)

[4.3.1 Creating API with FastAPI 73](#_Toc104279643)

[4.3.2 The pattern of resource, resource instance and related resource 74](#_Toc104279644)

[4.3.3 API architecture 76](#_Toc104279645)

[4.4 Technical documentation 79](#_Toc104279646)

[4.4.1 Low-level documentation 79](#_Toc104279647)

[4.4.2 High-level documentation 80](#_Toc104279648)

[4.5 Minimalist user interface 81](#_Toc104279649)

[4.6 Containers with Docker Compose 85](#_Toc104279650)

[4.7 The necessary output 88](#_Toc104279651)

[4.8 Summary 89](#_Toc104279652)

[5 Conclusion 91](#_Toc104279653)

[References 94](#_Toc104279654)

# Abbreviations

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

# Figures

[Figure 1 - Example of administration process of an aplication for approval of the proposal of the topic of the final thesis 18](#_Toc104061687)

[Figure 2 - An example of RPC API with HTTP protocol implemented by Slack 26](#_Toc104061688)

[Figure 3 - An example of a query operation type and response with GraphQL 27](#_Toc104061689)

[Figure 4 - Relational database to store phases and transitions 33](#_Toc104061690)

[Figure 5 - The interactive documentation system auto created by Swagger UI from openapi.json 35](#_Toc104061691)

[Figure 6 - Phases and transitions visualization with CytoscapeJS 37](#_Toc104061692)

[Figure 7 - Containerization vs Virtualization. 38](#_Toc104061693)

[Figure 8 - Docker network design based on CNM 38](#_Toc104061694)

[Figure 9 - Visualization of blob, tree and commit 41](#_Toc104061695)

[Figure 10 - Visualization of a commit history 42](#_Toc104061696)

[Figure 11 - Illustration of a linear administration process. 45](#_Toc104061697)

[Figure 12 - Illustration of a non-linear administration process 47](#_Toc104061698)

[Figure 13 - Administration processes with directors 48](#_Toc104061699)

[Figure 14 - Form structure 50](#_Toc104061700)

[Figure 15 - ER Diagram cardinality 53](#_Toc104061701)

[Figure 16 - Data model of user, group, role, and position 55](#_Toc104061702)

[Figure 17 - Simpler (left) and extended (right) solution for data model of administration process 57](#_Toc104061703)

[Figure 18 - Phases before and after transformation 57](#_Toc104061704)

[Figure 19 - Shifting relationship between forms and sections to phases and sections 58](#_Toc104061705)

[Figure 20 – Data model of administration process 59](#_Toc104061706)

[Figure 21 - Data model of the instance 60](#_Toc104061707)

[Figure 22 - Adding unique constraint for table contents explicitly in PostgreSQL 62](#_Toc104061708)

[Figure 23 - Creating table directors and handlers with unique constraint on multiple columns in PostgreSQL 62](#_Toc104061709)

[Figure 24 - Creating an envelope 64](#_Toc104061710)

[Figure 25 - An example of two commits 65](#_Toc104061711)

[Figure 26 - Data model for auditing 66](#_Toc104061712)

[Figure 27 - The result of the query to get full instance 67](#_Toc104061713)

[Figure 28 - The result of the query to get potential directors or potential receivers 70](#_Toc104061714)

[Figure 29 - The result of the query to get full instance in a commit 72](#_Toc104061715)

[Figure 30 - Interactive documentation created with Swagger UI 74](#_Toc104061716)

[Figure 31 - Class diagram for creating API with FastAPI 77](#_Toc104061717)

[Figure 32 - An example of low-level documentation for an endpoint 80](#_Toc104061718)

[Figure 33 - Creating phase in user interface 82](#_Toc104061719)

[Figure 34 - An instance example with the user interface 83](#_Toc104061720)

[Figure 35 - Transiting the instance in the user interface 84](#_Toc104061721)

[Figure 36 - The commits history (audit trails) example of an instance 85](#_Toc104061722)

[Figure 37 - Bridge network created by Compose 86](#_Toc104061723)

[Figure 38 - An example of build folder created by React 86](#_Toc104061724)

[Figure 39 - Communication between client and containers 88](#_Toc104061725)

# Tables

[Table 1 - Data types of data model 54](#_Toc104061726)

[Table 2 – 6 types of generic endpoint 75](#_Toc104061727)

# Introduction

## Goal and assignments for this thesis

Our goal is to create an information system (with access via web) with functionality supporting electronic administration process with elements enabling an audit of the performed tasks. From this goal we must specify which functionality our system must support for electronic administration process, and which performed tasks are audited. The requirements for the system on functionality supporting electronic administration process are as follows:

* It must be possible to define a sequence of phases (administration process) by experienced user with proper authorization.
* It must be possible to define who is/are responsible for every phase in administration process.
* It must be possible to define a structure of form by experienced user with proper authorization.
* Applicant must be able to create an instance from defined form (hereafter referred to as the instance).
* Instance must be able to transit through administration process.

The performed task in administration process is the transition of instance through phases. The creation of process, assignment responsible person/people for each phase, creation of form is not considered performed task in administration process. It is considered as the creation of administration process. The requirements on audit of the performed tasks are:

* In each phase of administration process, the instance must be recorded what is changed in the content, who changed it, when it is changed.
* It must be possible to recover instance at any phase.

The assignments of this thesis are:

* Design the information system as a set of cooperating containers (docker compose).
* Study API creation (according to the OpenAPI / Swagger standard) using the FastAPI library.
* Create an information system data model.
* Implement API.
* Prepare technical documentation for API.
* Create a minimalist user interface for the ability to demonstrate functionality.

The necessary output is:

* Source code (on the GITHUB platform) of the container providing the service to authenticated users.
* Documentation for the API (application interface) through which the service is available.
* Source code (on the GITHUB platform) of the container implementing the user interface.

## What is administration process?

In this thesis we can understand an administration process is a collection of phases, where a request of an applicant must traverse through. In each phase, responsible person/people will handle the request of the applicant, and then send the request to the next phase, where other person/people will again handle it and send it to the next phase. The process will continue till when the request is completely handled.

Below is an example of an administration process, which is based on Study order of the Dean of the faculty of military technologies, University of Defense in Brno, specified how student propose his / her own topic of the final thesis:

* The student submits the application for the topic of the final thesis to the head of the department for comment (1). The head of the department in accordance with the Rector's Measure No. 3/2018 Final Thesis will assess the fulfillment of requirements for the proposed topic of the final thesis, goal, description of the issue and justification of the topicality of the topic to the submitted proposal.
* With the opinion of the head of the department (2), the student will deliver the application to the Study Group of the Dean's Office of the Faculty (3).
* The Vice-Dean for study and pedagogical activities will request the opinion of the guarantor of the study program on all Applications for student’s own final thesis topic (4). In case of non-approval of the proposed topic or other facts contained in the Application, the guarantor is obliged to state specific reasons for non-approval.

We can visualize the process by Figure 1:

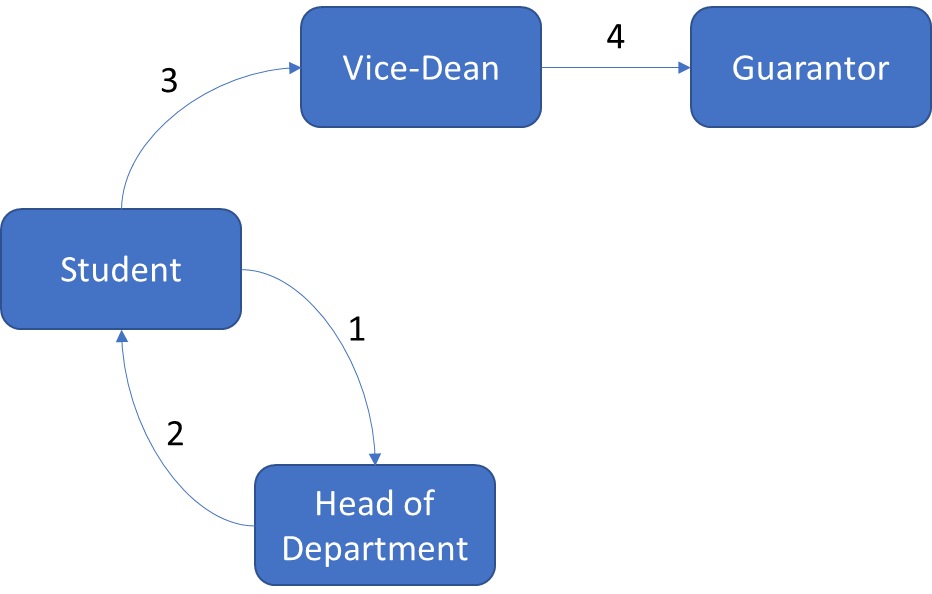


Figure 1 - Example of administration process of an aplication for approval of the proposal of the topic of the final thesis

Source: self-drawn

In general, the process will be defined by a regulation and be accompanied with a form. Applicant will instantiate an instance from this form and send it to the person in next phase. We will discuss about administration process more detail in later chapter.

## What is audit and audit trail?

An audit is often defined as an independent examination, inspection, verification, or review. The subject of auditing may be various in various sector.

The term audit is originally used in financial sector. Anthropologists have found records of auditing activity dating back to early Babylonian times (around 3000 BC). There was also auditing activity in ancient China, Greece, and Rome. The Latin meaning of the word “auditor” was a “hearer or listener” because in Rome auditors heard taxpayers, such as farmers, give their public statements regarding the results of their business and the tax duty due. The practice of modern auditing dates to the beginning of the modern corporation at the dawn of the Industrial Revolution in 18th century with the growth of the joint stock companies and the ownership and management became separate [1]. Audit is defined by Prof. L.R. Dicksee as an “*examination of accounting records undertaken with a view to establish whether they correctly and completely reflect the transactions to which they relate*”

In information technology sector, the definition of audit does not constrain nor presume the subject to which an audit applies. The International Organization for Standardization (ISO) defined audit in Guidelines for auditing management systems as: *“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 and CNSSI defined audit as: *“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 also defined audit trail as: *“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].

In case of administration process, we have determined that the subject of audit is the request of applicant, and the audit trail is chronological record of applicant’s request in each phase of administration process, that we can reconstruct and examines the sequence of phases, which the request traversed through in predefined administration process.

## Thesis’s structure

In chapter 2 we will define what is API, Web API, and information system with access via web. We also introduce 3 types of Web API and discuss about the advantages and disadvantages of these 3 types. The idea about resource of REST will be applied to implement the API along with URL, HTTP and JSON in this thesis.

In chapter 3 we will provide an overview about technology stack, which have been used for development our information system. Those technologies include PostgreSQL as a relational database management system, FastAPI as library (framework) for creating API, React as framework for creating user interface, Docker as virtualization (containerization) technology, Git and GitHub as version control system and source code hosting platform. Beyond these technologies, we find an inspirational idea from Git, and we will introduce it in the subsection The inspiration from Git.

Chapter 4 is where we discuss in more detail about administration process. Each requirement described in 1.1 will be clarified in section 4.1. Implementation of assignments will be described throughout section 4.2 to 4.6. The necessary outputs will be provided in appendix and the link to GitHub platform (detail in section 4.7).

The Conclusion summarizes the thesis, the benefit and limitation of the thesis and the orientation of our system for development in the future.

# Web API

## Application Programming Interface (API)

According to Bloch J., the idea of application programming interface arose from the idea of subroutine library, when Maurice Wilkes and David Wheeler created programs running in an early computer named EDSAC in the 1940s [6]. The key idea of subroutine and how it is documented is presented in 1952 in Wheeler’s paper *The Use of Sub-routines in Programmes*, in which he said: *“…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”* and *“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 is a set of definition and protocol on how two computer programs communicate to each other. API is not the implementation but the specification. In a simple word, a computer program takes an input, process it, and produce an output, then API describe how an input should be, how to feed the input to the computer program and how the output should be. For example, A CPU with its instruction set has an API, an operating system with a set of system call has an API, a programming language with a set of standard library functions or built-in functions has an API.

The first key idea of API is the concealment of internal implementation of a computer program, so that it can be reused or combined easily by another program or programmer. The second key idea of API is the ability of independent reimplementation of computer program, so that it does not impact on another program, which is dependent on it, despite of any purpose of the reimplementation, either to upgrade or to degrade it.

A computer program, which implements an API and provides it for another computer program to use, is called API provider. A computer program, which consumes the API, is called API consumer. API provider and consumer can also be referred to programmer, as programmer is the one who writes computer program for providing or consuming of another computer program.

## Web-based application software and Web API

An application software is a computer program, which solves end-user’s need for a specific task, which also can be solved without applying this program. Writing a document can be solved with pen and paper, but it also can be solved with word processor. Accounting and auditing can be solved with pen and paper, but it also can be solved with an accounting and auditing software. Administration process can be solved with pen and paper, but it can be solved with an application software, occasionally web-based software, which we have tried to solve in this thesis.

A network-based application software is an application software, which exchanges its data throughout the network. A more specific type of network-based application software is web-based application software (or simply web app), which is created from technologies underlying the World Wide Web (or simply Web) e.g., URI, HTTP, HTML, … A web app implementing an API is called web API.

## Information system with access via web

An information system (IS) is a set of interrelated elements or components that collect (input), manipulate (process), store, and disseminate (output) data and information and provide a corrective reaction (feedback mechanism) to meet an objective. Processing can be done manually or with computer assistance. A computer-based information system is a single set of hardware, software, databases, telecommunications, people, and procedures that are configured to collect, manipulate, store, and process data into information [7]. An application software is the nearest part of software element to the end user of the information system. Other parts of software element include operating system, networking software, …

As defined in 1.1 the IS must be accessed via web, therefore the application software must be a web app. The web app is also required to implement an API and its technical documentation as specified in the assignments of this thesis.

In next sections we examine 3 most popular Request-Response type or style of designing a web API [25].

## REST and RESTful API

### REST

Representational State Transfer (REST) is introduced in chapter 5 of the dissertation of Fielding R.T. in 2000 as an architectural style for distributed hypermedia systems [5]. REST is used for guiding the design and development of the architecture for the modern Web. Fielding described REST as a set interaction constraint between components (e.g., client-server, stateless, cache, uniform interface, layered system, code-on-demand), which he calls the style, with the regard to engineering principles (e.g., separation of concerns principle, principle of generality) to derive desired architectural properties of a modern Web architecture (e.g., performance, scalability, simplicity, modifiability, visibility, portability, and reliability).

Fielding R.T also described the experience and lessons learned from applying REST while authoring the Internet standards for the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI) in chapter 6 of his dissertation. These two specifications define the generic interface used by all component interactions on the Web.

### RESTful API

In the Web development community, a Web Application Programming Interface (Web API or Web Service) conforming to the REST architectural style is referred to REST API or RESTful API [27][28][29]. REST has been developed to represent the model for how the modern Web should work [5]. As a result, the rationale of REST is used to guild the standards or technologies in web, especially URI and HTTP.

Uniform Resource Identifier (URI) is used to uniquely identify a resource, which is the key abstraction of information in REST, throughout the Web. The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location") [8]. Because the term URL is a more specific term in context of this thesis, hereinafter we have used the term URL instead of URI. A resource of our IS on another hand can be any entity presented in our system e.g., applicant, a form, an instance, a phase, a transition, etc.

Hypertext Transfer Protocol (HTTP) is one of many communication protocols used in Web to exchange representation of a resource. It has methods e.g., GET, POST, PUT, PATCH, DELETE, as defined in RFC 2616 [9] to provide interactions (CRUD) with resource identified in URL. These methods are general enough for interaction with almost every type of resource. The exception is the interaction, which requires a specific parameter(s) than states of resource itself. For example, the transiting of an instance to next phase requires information about responsible person/people, which is not the state of the instance itself. Searching throughout a collection of resource also a typical problem with REST API. Although we can consider an interaction between resources like another resource as definition about resource in [8]: *“… abstract concepts can be resources, such as the operators and operands of a mathematical equation …”*, we will mix the concept about resource and interactions and cause the confusion and ambiguity for the programmers, as either a maintainer or a consumer.

A data format (media type) of representation of a resource in REST API is often, but not always, JSON or XML[[1]](#footnote-1). In this thesis we have used JSON as data format of representation of resource because of its more simplicity in syntax and less overhead than XML.

Applying REST style in design of API are a controversial topic [22][23][30], where REST APIs must be hypertext-driven. Fielding explains, that *“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].

The limitation in time and human resource requires more effort to apply the REST in the design of our API to be RESTful. To accelerate the process of studying creation of API and implementing API, the API design will not follow REST constraint in using hypertext. However, the idea about resource will be applied. URL, HTTP and JSON will also be used to create the web API in this thesis.

Pros:

* Easy to implement, maintain and capable to scale up
* Suitable for CRUD resource and showing relationship among resources

Cons:

* Not suitable for specific action of a resource
* Big payload in response
* Multiple HTTP round trips is required to retrieve representation of multiple resources.

## RPC

Remote Procedural Call (RPC) is a type of API implementation, in which a client program calls a function whose implementation is in a remote server [25]. RPC-style APIs focus on action instead of resource like RESTful API. Each endpoint represents a function implemented on the server that the client can call. Similar to a normal function, the endpoint receives parameters and returns a value that is sent as a response to the client. For an RPC API that uses HTTP as the transport protocol, the method or function can be placed in the URL and the arguments either in the query string or request body.

RPC style is suitable for APIs that expose a variety of actions of resources, which might have more nuances and specific than CRUD actions. RPC has partially solved the problem of REST, but it also has its own problems. Every type of resource has some common action and some own specific action. Firstly, we repeatedly implement the common actions on every type of resource. Secondly, it is not ideal to standardize, manage, maintain, or consume with such diverse list of actions, cause that list will expand over time when more actions or more types of resource with its own specific actions are added.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2 - An example of RPC API with HTTP protocol implemented by Slack

Source: <https://api.slack.com/methods>

Pros:

* Suitable for specific action on a resource

Cons:

* Only apply for specific solution, lack of standardization
* Can lead to function explosion if more action is added

## GraphQL

GraphQL is a query language for APIs, which was developed internally by Facebook in 2012 and was publicly released in 2015 [10] and has been adopted by API providers like GitHub, Yelp, and Pinterest.

GraphQL allows consumer to define the structure of the data required, and the server returns data with exactly that structure, thus reduce the redundant data in response, reduce number of HTTP roundtrips in the comparison with REST API, is suitable for resources with a large amount of data field in its state and for the resource in high hierarchical relationship with other resources. GraphQL has a single endpoint to interact with and two type of operations, query for read-only fetch data and mutation to write followed by a fetch.

Graphical user interface, text, application, Word

Description automatically generated

Figure 3 - An example of a query operation type and response with GraphQL

Source: <https://graphql.org/learn/queries/>

The drawbacks of GraphQL are the complexity in the implementation, additional processing to parse queries and verify parameters. Therefore, GraphQL is not suitable for a simple API. Nevertheless, GraphQL can be used in combination with REST API, and it can be considered when system become larger. For simplicity of our system in implementation, we have not used GraphQL in this thesis, but it is considered a potential solution when the system evolves and becomes more complex.

Pros:

* Less redundant data in payload of response
* Less HTTP roundtrip

Cons:

* Additional complexity
* Additional processing
* Not suitable for simple API

## Other type of web API

Other type of event-driven web API e.g., Webhook for server-to-server communication, WebSocket for bidirectional client-server real-time communication, HTTP Streaming for sending an infinite chunk of data in infinite numbers of response; are considered unnecessary for the requirement of our IS, so they are not further discussed in this thesis. Nevertheless, it is possible to combine these type of web API with other types of web API if more requirements on our IS are made.

## Summary

In this chapter, the topics have been discussed are: what is API; what is web API; what IS with access via web; three types of web API mostly used in web development, those are REST API, RPC API and GraphQL API. We have considered pros and cons of each type of web API, and I as the author of this thesis choose to use REST as a guild line in the design of our web API.

In the next chapter we will examine technologies stack used for development our IS in this thesis.

# Technology stack

In this chapter we consider which technologies will be used in the development for our IS and why we choose them. Because there is no requirement on which database will be used, firstly we examine types of databases and I as author of this thesis will choose the type of database according to the ability to solve the requirements on the IS, time limitation of the thesis, my own ability and experience with database. Second assignment is to create API using FastAPI library, hence we will introduce the FastAPI and additional tool using with it. Next, we will introduce React as technology for creation a minimalist user interface for the ability to demonstrate functionality as being assigned in section 1.1. We choose to introduce Docker in the last section, because it only has meaning when other software preexists, even when the first assignment for this thesis is about Docker.

## Types of databases

Two requirements we must consider when we choose a type of database is the possibility to define administration process and the possibility to define a structure of form. An administration process is a sequence of phases and transitions, or nodes and edges in term of graph theory. A form contains a varying number of fields, which create the structure of the form. Our system must be able to store such administration process and such arbitrary structures of different forms, so that users are able to initialize an administration process and instantiate an instance from corresponding form and fill data to it. As a result, our system must also be able to store current phase of the instance and data filled by user. To fulfill these requirements, we consider two types of modern databases, and those are relational and non-relational database, also known as SQL or NoSQL database. Then we will choose one type of database and its specific instance to use in our system.

### Relational database

Relational database was introduced by E. F. Codd at IBM in 1970 in his research paper *A Relational Model of Data for Large Shared Data Banks* [16]. A relational database stores data in relations, which the user perceives as tables. Each relation is composed of tuples, or records, and attributes, or fields. The physical order of the records or fields in a table is completely immaterial, and each record in the table is identified by a field that contains a unique value (also known as primary key). The relational model categorizes relationships as one-to-one, one-to-many, and many-to-many. A relationship between a pair of tables is established implicitly through matching values of a shared field (also known as foreign key). Tables and their relationships form a schema of a database. Relational database assumes a well-defined structure in data and schema. It also assumes that indexes can be consistently defined on data sets and that such indexes can be uniformly leveraged for faster querying [31]. As soon as these assumptions do not hold true, the database starts to show its disadvantages. One of these disadvantages that we must consider is the unstructured data, as our form and instance is a type of an unstructured data, where each form can have an arbitrary number of fields and an instance of a form must follow the form’s structure to store the data to it. However, with normalization, which is introduced by Codd in his first and second paper [16][17], we can tackle the problem of the unstructured data and in addition reduce the data redundancy, the need for redefining database schema (restructuring), anomaly operation while keeping consistency on data we store.

A relational database management system (RDBMS) is an application software, which provide an API and possibly a user interface, which allow us to create, maintain, modify, and manipulate a relational database. The most common way of interacting with relational database systems is using Structured Query Language (SQL), which allows for specific queries to be built for creating tables, searching, and filtering data across one or multiple tables. Some of the most popular RDBMS are Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite, and MariaDB.

### Non-relational database

A non-relational database, also known as NoSQL (can be understood as No SQL or Not Only SQL), is any kind of database that does not use the tables, fields, and columns structured data concept from relational databases. There are four types of NoSQL databases [12]:

#### Document database

Document database stores data in documents, which are typically in JSON-like structure that supports a variety of data types [31]. These types include strings; numbers like int, float, and long; dates; objects; arrays; and even nested documents. The data in document is stored in key/value pairs. A collection is a group of documents having similar content structure. Not all documents in a collection are required to have the same fields because of flexibility in schema of document database.

Retrieving data from document database varies between document databases and the solution is often proprietary. For example, to get all documents from collection users:

In MongoDB query takes the form of method chaining and data can be queried:

db.users.find({})

ArangoDB use its own query language and data can be queried:

FOR doc IN users

RETURN doc

These are equivalent in SQL as:

SELECT \* FROM users

Some of the most well-known document databases or document database services are MongoDB, ArangoDB, CouchDB, Amazon DynamoDB, Google Cloud Firestore, …

#### Key-Value database

Key-value database (also known as key-value store) stores data in a “key-value” pair and is optimized for reading and writing data by the key. The key uniquely identifies data and for each key, there is exactly one value, which can be simple data types like strings and numbers or complex objects.

This type of database implements a hash table data structure (also known as dictionary, map, hash map, symbol table), that is the key is hashed to a numerical value (the hash) and the value is stored in position defined by the hash. The value is retrieved by the key in constant time.

Key-value stores does not have query language like SQL. Values cannot be queried or searched upon, but only the key can be. Data is written (inserted, updated, and deleted) and queried based on the key.

Redis and Riak is typical examples of key-value store.

#### Column-Oriented database

Column-Oriented database stores data by column rather than by row as relational database. In relational database each row contains values of all columns related to all attributes of an entity instance. The table is a sequence of rows, and each row is stored as a block in disk. A row as a block is fetched with all its columns. If we want to retrieve a column or a small groups column of all rows, all block of table must be fetched. Therefore, it is more efficient to read data in such fashion if we store data related to an instance per column in table.

In column-oriented database each row contains data of an attribute related to all instances and is stored as a block in disk. An attribute or set of attributes of all instances is retrieved by fetching one block or a set of blocks without fetching whole table. To effectively query data in column-oriented databases, each unit of data in a block is a set of identifier/value pairs. Column-oriented database is fast in aggregate operation for example average age of user. Adding attribute is also fast by adding just adding another block on disk. Adding instance is slow because each attribute of all instances is continuously store in disk.

SQL can be used as query language in column-oriented database.

Some typical examples of column-oriented databases are Apache Cassandra, HBase.

#### Graph database

Graph database has solved the limitation of relational databases in schema by treating relationship as data. The data of entity instance is stored in nodes and the relationship is store in edges. Each node is categorized by adding label to it, as an entity instance is categorized in a table of relational database. Node stores attributes (properties) in form of key/value pair. Edge also has label, possibly attributes (properties), and additionally direction. The relationship is added by simply adding a new edge without redefining database and data migration.

Graph database is suitable for dealing with highly interconnected entities and is optimized to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL.

Graph database like Neo4j, which is one of the most well-known graph databases, uses Cypher Query Language to retrieve data from graph. It was inspired by SQL and has similarity with SQL [26].

### Choosing database

Consider the first and the third requirement on our IS, those are possibility to define phases and transitions of an administration process and possibility to define form structure for such process.

The phases and transitions are more like a nodes and edges in graph theory. Each phase in process can point to another phase, and the transition is created to indicate that connection. For different administration process it must be different phases and transitions. To not be confused with the graph database, graph database is used for highly interconnected entities and the relationships between entities are prone to change. Phase and transition are only two entities in our system and the relationship between them is not prone to change. We can easily store phases and transitions in two tables as the example shown below. With these two tables we can easily describe any administration process, including add, modify, or delete transition between two phases. There is no need to use a graph database for fulfilling this requirement. We will extent this topic in relationship between these two tables in chapter 4.

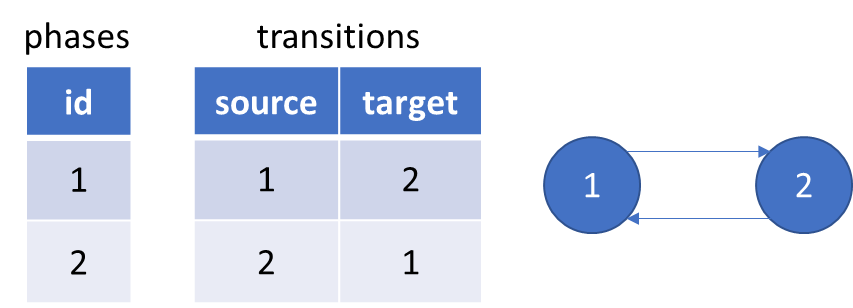


Figure 4 - Relational database to store phases and transitions

Source: self-drawn

The form is the structure for its instance. The structure of the form defines that, in the instance how many fields are presented and can be filled by user. The instance can be instantiated from the form and the user can fills data to it. In the programmer’s point of view a form is like a class and an instance is like an object, instantiated from the class. The problem is that the structure of the form is not predefined by programmer like a table in database to store data filled by user, because this structure itself is the data and is defined by user, and we must store it. Then another user like an applicant wants to instantiate an instance and fill data to this instance, and we must store this data in a manner, that we can retrieve it and know which field it belongs to.

One possible solution is that we store this structure of form in a document database, in which a collection name forms will store all form’s structure. These documents can have different fields because of the nature of flexible schema of document database. The key can be the name of the field and the value can be any metadata describing this field in any data format. Instances will be stored in documents of another collection name instances, in which field’s name can be derived from one document of the collection “form” as a key, and field’s value is data from user, metadata of the field can be used to validate this value.

Another solution is to split the structure of form and the data filled by user and store them separately in different tables as normal data. This solution can be achieved by using relational database and normalization. This solution is chosen for solving the requirements of the thesis because it shows the possibility of fulfilling thesis’s requirements. In addition, the reason for choosing relational database in the development our information system is learning something entirely new, including design method and tool, is a barrier to adoption. It takes time to master a tool with proper design method. Therefore, with the ability of relational database to solve the requirements on the IS, time limitation of the thesis, and my own ability and experience with relational database, I, as the author of this thesis, have decided to choose relational database to develop our IS. The chosen RDBMS to use in this thesis is PostgreSQL. Further discussion with relational database to solve thesis’s requirements has been made in chapter 4.

## FastAPI

### FastAPI

As introduced in the documentation of [FastAPI](https://fastapi.tiangolo.com/), it is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

FastAPI is based on and fully compatible with OpenAPI. It automatically generates a specification for all defined API conformed with the OpenAPI standard. The specification includes paths, the parameters they take, etc. The file for this specification is in JSON data format (also known as JSON schema) with name openapi.json. This file is used for interactive documentation system like Swagger UI to visualize and interact with the API’s resources.

Graphical user interface, text, application

Description automatically generated

Figure 5 - The interactive documentation system auto created by Swagger UI from openapi.json

Source: self-made

### SQLAlchemy

In Python the most popular database drivers for PostgreSQL is psycopg2. But to simplify the development process we use SQLAlchemy, a Python SQL toolkit and Object Relational Mapper (ORM). ORM can be used to create or delete table as DDL and is used mainly to insert, retrieve, update, delete data as DML, but with a higher-level abstraction. With ORM, we can process data in OOP paradigm in Python and without using SQL (although using SQLAlchemy ORM still requires the knowledge about SQL).

### Swagger specification and OpenAPI specification

The OpenAPI Specification was originally based on the Swagger Specification (version 1.0 to version 2.0), donated by SmartBear Software. In 2015 Linux Foundation announced the Open API Initiative, which extend the Swagger 2.0 specification and rebranded it as OpenAPI specification [11].

Swagger specification and OpenAPI specification are used to describe and document RESTful APIs. An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases.

An example of documentation generation tool is Swagger UI, which is used in FastAPI.

## React

Another assignment for this thesis is to create a minimalist user interface for the ability to demonstrate functionality. To complete this assignment, we use React and CytoscapeJS

React is a JavaScript (JS) library for building user interfaces. It can be used to create a single-page web app, which allows user to request a web page (HTML, JS, CSS, …) only once and this page using JS code will request the data from API and update the DOM tree. It creates a virtual DOM used for comparing and updating the DOM tree, which will be rendered to the browser. The process of comparing and updating the DOM tree is called reconciliation. React implements a heuristic diffing algorithm in reconciliation based on two assumptions:

* Two elements of different types will produce different trees.
* Child elements may be stable across different renders with a key prop.

These two assumptions help React reduce the time complexity from O(n3) to O(n) in a comparison with generic solutions to generate the minimum number of operations to transform one tree into another [15].

Another consideration when building a user interface for our system is the visualization of an administration process for user, which we will implement by using CytoscapeJS. It is a library written in pure JS, which allows us to visualize our phases and transitions for administration process as directed graph of nodes and edges as show in the Figure 6.

Chart

Description automatically generated with medium confidence

Figure 6 - Phases and transitions visualization with CytoscapeJS

Source: self-made

## Docker

### Docker and container

Docker is an OS-level virtualization software (often refer to containerization), which packages an application and its dependencies into a software unit, called container. It reduces the waste of resource e.g., CPU, RAM, storage, OS license for each application software with virtualization technology by using shared host OS for all containers run on it. It also provides a way for quicker build, test and deploy without platform compatibility barrier [13][14].

Chart, treemap chart

Description automatically generatedGraphical user interface, application

Description automatically generated

Figure 7 - Containerization vs Virtualization.

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

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker images (shortly image) is a lightweight, standalone, executable package of software that includes code, runtime, system tools, system libraries and settings, which are required to run an application. Images become containers when they run on Docker Engine at runtime.

### Networking in Docker and Docker Compose

Docker network design is based on Container Network Model (CNM), which contains 3 main components: Sandbox, Endpoint and Network [13].

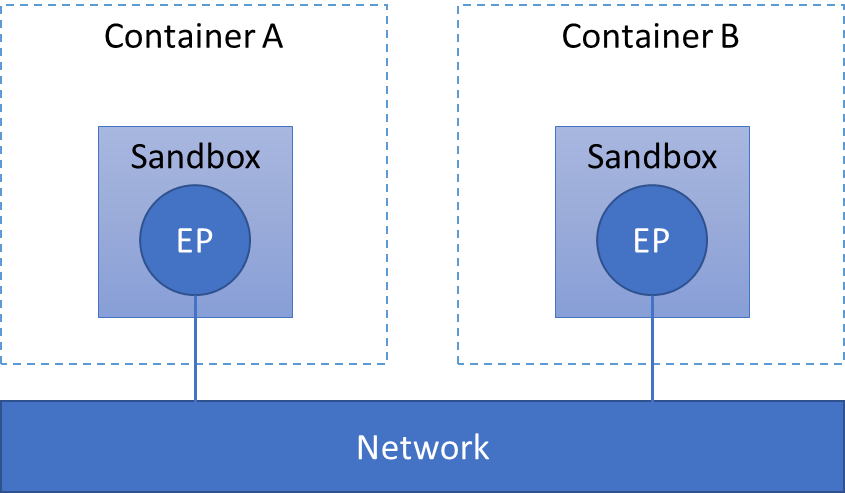


Figure 8 - Docker network design based on CNM

Source: self-drawn

* A Sandbox contains the configuration of a container's network stack. This includes management of the container's interfaces, routing table and DNS settings. A Sandbox may contain many endpoints from multiple networks.
* An Endpoint connect a Sandbox to a Network as a virtual network interface (e.g., veth). An Endpoint can belong to only one network, and it can belong to only one Sandbox, if connected.
* A Network is a software implementation of a switch, which group Endpoints together and enable them to communicate with each-other directly.

CNM is implemented by libnetwork and driver. The libnetwork provide an API to for user like Docker or third party to create and manage CNM objects for example Network or Endpoint object. The actual implementation of creation and manage these objects by user like Docker or third party is called driver. Several drivers exist by default in Docker, and provide core networking functionality e.g., bridge, host, overlay, ipvlan, macvlan. Other network driver plugins are provided by third-party in Docker Hub.

When we start Docker, a default bridge network is created automatically, and newly started containers connect to it. Containers running on the same host can communicate with each other in a default bridge network by using only IP addresses (not by the container name). A user-defined bridges in another hand provide automatic DNS resolution between containers and allow containers in the network to communicate by container name.

Docker Compose (shortly Compose) is a tool for defining and running multi-container Docker applications as services, which communicates with each other. By default, Compose sets up in a single host a bridge network for our containers and dynamically assigns IP address for each container. Each container joins the default network and is reachable by other containers on that network by container name defined in Compose file (usually in YAML format). The IP address is automatically allocated for each container by default in range 172.17.0.0/16. If we make a configuration change to a service and run docker‑compose up to update it, the old container is removed and the new one joins the network under a different IP address but the same name. Therefore, it is more effective to use the container name in the communication between services in the default network for either development or production rather than trying to configure which IP address of the container is.

Note that bridge network is a private network restricted to a single Docker host and all containers on this network can only communicate with each other through the bridge (virtual switch) created by the bridge driver. To receive and transmit traffic outside this network we must use port mapping. Docker allows a port on container to be mapped to a port on host. Any traffic with configured port come to the host will be directed to the container and enable the container to communicate with the outside world.

Our information system is a set of cooperating containers using Docker Compose as first assignment for this thesis. We will use Docker and Docker Compose in development of our IS. The structure of cooperating containers will be presented in the next chapter, after we present all other services.

## Git and GitHub

Git is a distributed version control system designed to handle from small to very large projects with speed and efficiency. Git was developed by the Linux development community (and particularly Linus Torvalds, the creator of Linux), while they developed the Linux kernel in 2005 [19].

GitHub is a source code hosting platform for version control and collaboration. It allows us to develop on a project with other developer from anywhere. It offers the distributed version control and source code management functionality of Git, plus its own features.

Beyond the purpose of Git and GitHub, which are used for controlling the version of our source code, we see an inspirational idea from Git that can be applied in our system, which was introduced in the next subsection.

### The inspiration from Git

Git thinks of its data more like a series of snapshots of a miniature filesystem. For each file in our project, Git will store the file’s content in another file in its own database (in the .git/object folder) with a hash (using SHA-1) as the name of the file. Git call this type of file a blob object. Git use tree object to store either the filenames and the hashes of a group of blobs, or the directory name and the hash of another tree(s). The tree object is like a directory in UNIX file system and the blob is like the file. When we commit, Git generates a top-level tree from the file structure of our project, which is the “snapshot” of our project. It compares the hash of each file and if the files have not changed, the hash will not change either and Git does not store the correspond blob again, just a hash to the previous identical blob, which has already stored. Git then use the commit object to store the hash of top-level tree, the hash previous commit if any, the author/committer information with a time stamp and the commit message. Figure 9 is an example of the blob, tree, commit from the book *Pro Git* written by Scott Chacon, Ben Straub.

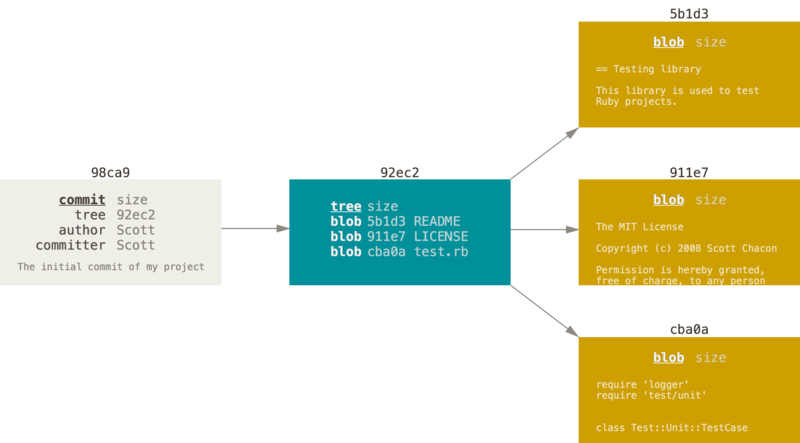


Figure 9 - Visualization of blob, tree and commit

Source: [19]

Other approach different from Git is to store the based files and the differences of their content made over time (often refer to delta-based version control). With this approach, if we want to retrieve a file in a particular commit, we must compute from the based file and add all its delta up to the specified commits. Moreover, Git approach make branching and merging commits a lightweight, fast, and very simple operation in comparation with other VCS [19].

Next, each commit will link to the previous commit through the hash of previous commit and create a history of commits. If we do not consider the mechanism of branching and merging the commits provided by Git, the commit history will be linear and we can consider this commit history as chronological records of our project, and we can reconstruct and examine the project at any commit in the history of the commits.

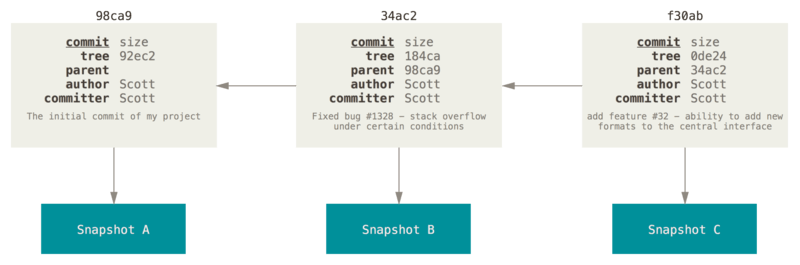


Figure 10 - Visualization of a commit history

Source: [19]

If we think about auditing an applicant’s request as we introduced in subsection 1.3, we only need a linear history of the request through a sequence of the phases, because a history of a request cannot be branched and merged like the source code of a project, where multiple developers collaborate and modify the source code in parallel. The request can only appear only at one phase of the process as in administration process with paper, where one sheet of paper cannot exist in two or more phase. Regardless to the mechanism of branching and merging commits provided Git, we use this approach to solve the requirements in this thesis because of its simplicity and robustness. We will apply this idea in a simpler manner and adjust it to achieve the goal of our system. The more detail description about our adjustment is provided in subsection 4.2.4.

## Summary

In this chapter, 2 types of databases are introduced, those are relational and non-relational database. Each type has its own pros and cons, and it is used in depend on specific use case. I as the author of this thesis decided to choose PostgreSQL as database management system to store data because of its ability to solve the requirements on IS, the time limitation of the thesis, and my own ability and experience with relational database. Other framework, library and technologies used to develop our IS are also introduced, those are FastAPI for creating API, React for creating user interface, Docker as containerization technology, Git as version control system and GitHub as source code hosting platform as specified in the assignments of this thesis. In the next chapter we will present the design of our IS and how we implement our IS using these technologies.

# Discussion and implementation

In this chapter we discuss about the problem domain and show how we have implemented the assignments described at the beginning of this thesis, to fulfill the requirements on information system, produce the necessary output and achieve the goal of the thesis.

The first section discusses about the administration process in detail. The following questions will be answered: How to define sequence of phases for an administration process? How to define who is/are responsible for each phase? What is a structure of form and how to define it in an administration process? How applicant create an instance from defined form and transit it through phases? What is the subject of audit? What is the purpose of auditing? When an audit trail should be made? To answer this question, we analyze some examples of administration process, generalize the terms, and draw out the characteristics of these examples.

These characteristics will be used in the data model of our system and imposing constraints for this model, which is described in section 4.2.

Section 4.3 describes about the design of API and its implementation with FastAPI. The pattern of API and how it is applied in API architecture is also described in this section.

Technical documentation is described in section 4.4, in which we divide documentation to a low and a high-level documentation.

A minimalist user interface created with React in section 4.5 will show how the instance, the instance’s transition and the instance’s commits history should look like as demonstration of functionalities of the API.

In section 4.6 a set of containers created with Docker Compose is presented in form of the bridge network of containers in a single host and of the communication between them.

Lastly, the necessary output for this thesis is described in section 4.7.

## Administration process

In an organization, an administration process is defined by a regulation as a sequence of phases and is usually accompanied with a form. An applicant instantiates an instance from the form, initiate the first phase of the process and traverse it through other phases (the case, where different forms is used in different phases, is considered splitting into different administration process, and therefore is not considered further in this thesis). The applicant's desire to resolve his/her own issue by instantiating an instance, initiate a process, and moving the instance through phases with the goal to achieve his/her desire is called a **request**. The term request is also used to indicate the instance with a particular sequence of phases, which the instance traversed through.

The responsibilities for different phases in a process are allocated to different entities e.g., branch, faculty, department, workgroup, study group, etc. we generalize the term and call them **group**. These responsibilities are specified and expressed further as a **role** in a group. The roles in groups often have the same pattern if the groups have the same level in organizational structure. For example, in a University each Faculty will have a Dean and Vice-Deans, each Department will have a Head of Department, Professors, Docents, Assistant Professors, ... There can be a case, where groups in the same level in organizational structure does not have the same role such as there is Deputy Head of Department instead of Head of Department. In both cases, to specialize the role related to a group we use the term **position** e.g., Rector of University of Defense, Dean of Faculty Military Technology or Head of Department of Informatics and Cyber Operations… are all positions. The position also can represent as a general role and is not related to any group. In this case position and role are equivalent and two terms are used interchangeably.

### Phases in administration process

A **phase** is separate part of a process, where user(s) with assigned position(s) handle section(s) of the instance and where an audit trail is made. The user handling the instance in a phase is the **handler** of that phase. There can be multiple handlers in one phase. “Handle” means that the instance will be modified some part and/or some actions can be made in real life e.g., checking the validity of information filled by applicant.

A process will begin with one phase and end with one phase, we call it **begin phase** and **end phase**. Other phases between these two phases we call **transit phase**. In practice there can be only one begin phase, only one end phase and multiple transit phases. The process can begin with two or more handlers in one phase but cannot begin with multiple phases. If a process designed to begin with one begin phase from multiple begin phases (or end with one end phases from multiple end phases), it should split into separate processes. If a process designed to begin with multiple begin phases (or end with multiple end phases) in parallel, it should be merge into one begin phase (or end phase) with multiple handlers. Let consider the example where an administration process defined in STUDY ORDER OF THE DEAN OF THE FACULTY OF MILITARY TECHNOLOGIES, Submission and processing of final theses, article 5 Processing the assignment of the final thesis and its approval [18]:

* The thesis supervisor and the student will prepare the Final Thesis Assignment on the prescribed form. Supervisor and student sign the form and student hands over the Final Thesis Assignment to the head of the department (HoD)
* The HoD will comment on the assignment and hand it over to the Vice-Dean for study and pedagogical activities (VDSPA)
* VDSPA submits all Final Thesis Assignments of faculty students to the guarantor, who will decide in writing on all Final Thesis Assignments
* After the approval of the Final Thesis Assignment, the Study Group of the Faculty Dean's Office will provide each Final Thesis Assignment with an official stamp. Subsequently, two copies of the Final Thesis Assignment are handed over to the student, one copy will be received by the supervisor of the final thesis and one copy will be placed in the student's file by the Study Group of the Faculty Dean's Office.

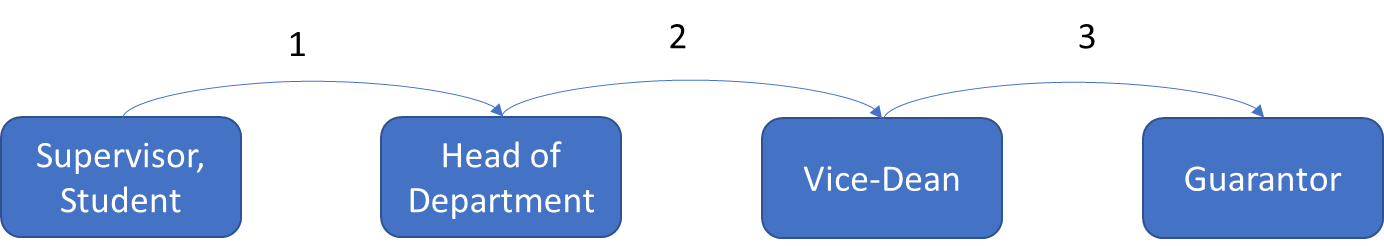


Figure 11 - Illustration of a linear administration process.

The numbers indicate the order of transition.

Source: self-drawn

We see that, there is only one begin phase handled by supervisor and student. Its purpose is to prepare the assignments for the final thesis. Next transit phase is handled by the HoD and its purpose is to achieve the comment of the HoD on the assignment. The instance is then transited to the next phase handled by the Vice-Dean and the Vice-Dean will gather all the instances of Final Thesis Assignment to the guarantor. Only one end phase is handled by Guarantor, who will decide in writing on all instances. The last point is just the providing the copy of the instance to the participants of the process, instance will not be further modified and therefore there will be no next phase and the instance end at the phase of Guarantor.

The above example only shows a linear process, where an instance is transited to the next phase without the possibility to turn back. As a result, it does not describe all the possible transitions in practice. For example, if HoD finds a typo of student, there is no transition back to the first phase. However, the transition is not described did not mean that it cannot be made. The HoD can inform the student about the typo and require the student to correct the instance and resubmit it. In this case the transition is implicitly made by the HoD and because it does not affect much to the process, it is not described in the regulation.

Another process defined in article 8 of [18], Opponent of the final thesis, which describes the transition back as follow:

* The HoD submits the proposal of the opponents of the final theses to VDSPA (1)
* The VDSPA will request the opinion of the guarantor on the opponents of the final theses (2)
* In case of disapproval of the proposed opponent of the final thesis, the guarantor will justify the disapproval and propose a new opponent of the final thesis. The guarantor may consult with the HoD on the proposal for a new opponent of the final thesis (2.1)
* The opinion of the guarantor on the proposed list of opponents of the final theses is submitted to the Dean through the VDSPA (3) (4). The Dean approves the list of proposed opponents of the final thesis. In case of disapproval of the proposed opponent of the final thesis, the Dean justifies the disapproval and proposes a new opponent of the final thesis. The new proposal of the opponent of the final thesis will be submitted to the guarantor for approval (4.1). The guarantor will forward his / her opinion to the Dean through the VDSPA (4.2) (4.3). The Dean will approve the new opponent of the final thesis.
* The list of defended final theses, including the supervisors and opponents of final theses, is published in the Dean's study order.

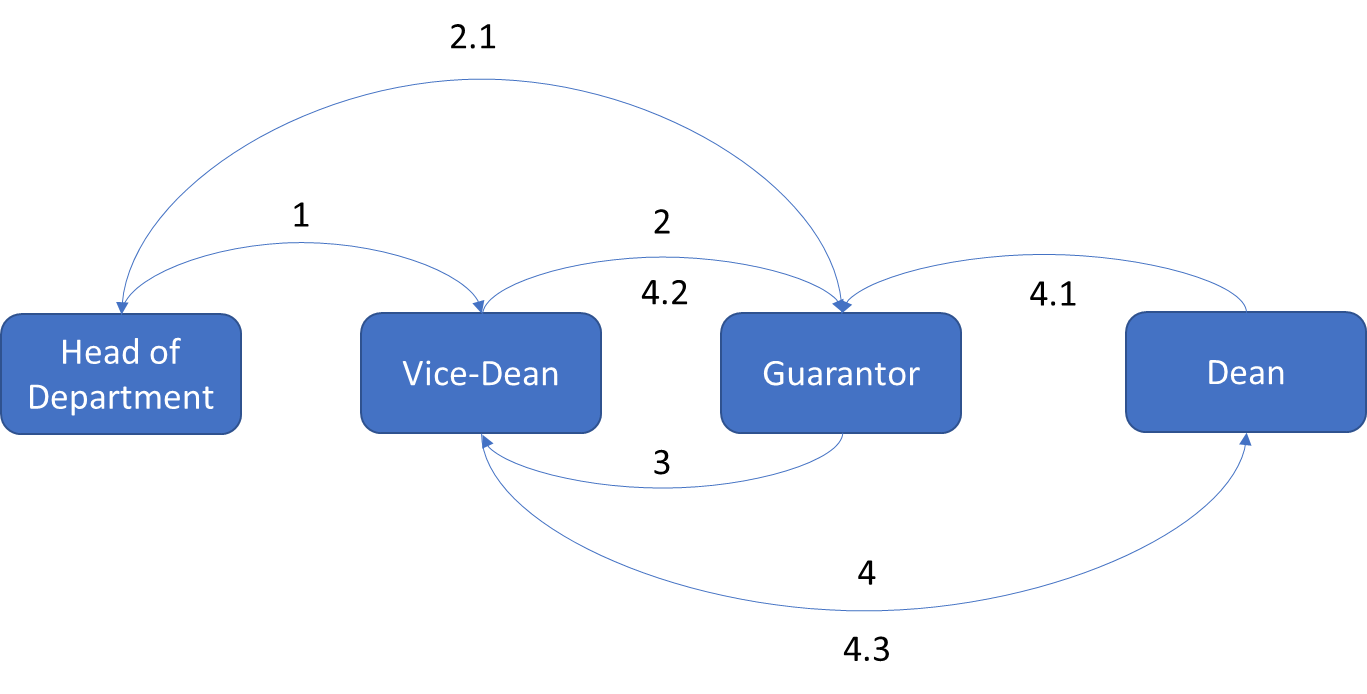


Figure 12 - Illustration of a non-linear administration process

Source: self-drawn

We can see that, transition (2.1), (4.1), (4.2) and (4.3) is a transition back, when the handlers disapprove the information in the instance. The request created from this process can go through this transition, but it is not required.

We also see that transition (2) and (4.2), (4) and (4.3) should have been different transitions, because the request goes through them with different purpose. However, the transition should only define this availability, whether a request can be transited from one phase to another without consideration of the purpose of the transition. The purpose of a transition can be varied depending on the request, and therefore should let the participants of the request determine. Defining multiple transitions from one phase to another separately is redundant and process designer may not catch all the cases happened in the transition between these two phases.

Notice that there is no transition from one phase to itself. In practice, the process is divided into phases because: firstly, each phase will be handled by different handler at different workplace; secondly, the completion of one phase is a requirement for the next phase to be handled. Therefore, if a process designed with a phase, which transit to itself, it should be only one phase. If the designer intends to create a process in such manner for the audit purpose the process should be split into two phases.

### The designated position and director

In both above examples, each phase can be handled by multiple handlers but there is only one position can make a transition. A such position is called **designated position** and the handler with designated position is called the **director** of the phase, because the director decides the direction of the instance in the process. For example, the director of the begin phase in first example is the student; the directors of all other phases are the handlers of those phases.

The designated position is not necessary one of the assigned positions of a phase and the director in such case only performs the redirection, not instance modification. Note that in both above examples, the VDSPA does not modify any field in the instance, but (s)he does make two actions: 1 is to check the validity of the information in instances, 2 is to redirect the instance to the next phase. In the first example, there is no direct transition from HoD to Guarantor. In the second example there is no direct transition from HoD to Guarantor, or from Guarantor to Dean. If we merge the phase of VDSPA to the phase of HoD in both examples, merge the phase of VDSPA to the phase of Guarantor in the second example, we can designate the VDSPA to become only the director and not the handler of these phases. The phases with director and handler of the above example can be visualized in Figure 13.

Graphical user interface

Description automatically generated

Figure 13 - Administration processes with directors

Source: self-drawn

### Structure of a form

A form defined for an administration process must contain fields and each field is assigned for only one position. The data filled in a field can be affect by multiple handlers but only one handler filled the field, for example the thesis supervisor and student can prepare the Final Thesis Assignment, however in the form, the assignment is only filled by the supervisor and not the student. To simplify the structure of the form we can group field(s), which is assigned for a position, together to a **section**. In other word section provide a functional grouping method for the form.

Section also provides a semantic method in grouping the fields. Handler will know the fields, which they must handle, are in one section and not in several section. Section therefore used to structure the form and helps the handlers know which fields they must handle and which fields they must not handle.

The presentation of the fields assigned for one position are usually arranged to one place in the form and rarely in separate places and we can also group these fields into one section. Section thus provides presentational method in grouping the fields. The presentation order of sections in a form will usually match the sequence of phase of correspond process, which the instance must traverse through. However, form is a linear document, and the order of section can only match the sequence of phase if the process is also linear. In case of non-linear process, the order of section cannot match the sequence of phase. The creator of the administration process and form will usually compensate this problem by trying to define the process and present sections in the form as most linear as possible.

For each phase, there are multiple sections, which could be handled, and each section can contain multiple fields. In practice fields of a section is filled by only person with assigned position. The structure of a form can be visualized in the figure below.

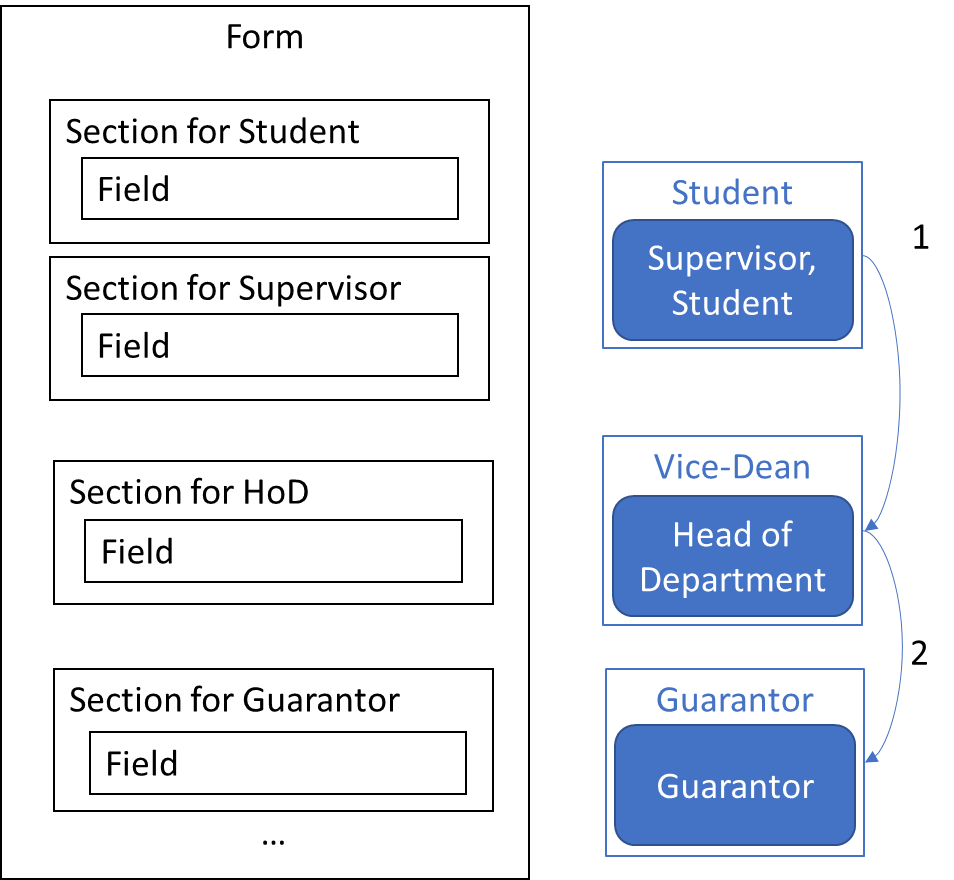


Figure 14 - Form structure

Source: self-drawn

### Instance and its transition

To create a request, applicant instantiate an instance from the form, which is defined for correspond administration process in regulation, and initiate the process. After filling all fields assigned for applicant, he transits the instance to the next phase of process and two case can be happened: the applicant knows or does not know who the handlers of the next phase are.

If the applicant belongs to the organization, the applicant should know and must know who the handlers of next phase are. For example, the student must know the head of department, which he belongs to, or the head of department must know the vice-dean of faculty, which his department belongs to. Such type of administration process we call interior process.

If the applicant does not belong to the organization but is a subject outside the organization, the applicant may not know who the handlers of the next phases will be. For example, a new student, who want to register to the university, does not know who the receiver of his request will be. We call this type of administration process an exterior process. It is a typical type of administration process in public sector, where the organizations resolve the issue of citizens. The applicant either sends the request to a representative of a group, which is assigned for handle such type of request, or presents at the organization and directly submits the request to the staff, who will handle it. Moreover, organization in public sector will have multiple handlers with the same position, usually the staff, to handle more requests at the same time. The requests will be put in a queue and the handler is undefined until the request is received. A coordinator, either a system or a person, will then coordinate handler for handling instances.

The information system in this thesis can handle these two cases. The only common requirement for these two cases is that the applicant must specify who is the director of the next phase. In the first case the applicant can specify who is the receiver of the next section(s). In the second case the applicant can leave the receivers of the next sections as unspecified and when the instance is transited to the next phase, the director of the next phase will specify the receivers for the sections. The director in the second case plays the role of a representative of the group and the role of a coordinator.

The director not only can specify the receivers for the sections but must also be able to change the receiver of a section. For example: the Head of Department chose a Leading Teacher for the Student but the Leading Teacher refuse, so the Head of Department must choose another receiver for the section of the Leading Teacher.

After the instance is received, the receiver becomes the handler of the section and can modify the fields assigned for him/her. The director of current phase can transmit the instance to the next phase at any time and the process will continue till the end phase. At the end phase, when the instance is completely handled by all handlers, the director can mark the instance as completely done and the instance cannot further be modified.

### Audit and audit trails

The subject of auditing can be varied, depending on what is the purpose of audit. We can audit the login of users to the system, action of user on system such as creating a group, change role, delete a position, creating a form… for security purpose. Or we can audit the joining and leaving of users on a group, the assigning and revoking of a position for users, publishing and deprecating a form…However to support the administration process, the most important subject we must audit is the request of applicant.

All effort of the applicant in a request is to achieve permission, directive, agreement, approvement, expression, statement, or opinion on his request from handlers at each phase and to reach the end of the process. We can say the applicant want to achieve his goal and the handler grant him achievement. In administration perspective, the end phase is where the request is evaluated at the highest level of administration. The handler at the end phase usually does not know who edited the request, at what time the request was edited, which phases the request was traversed through and what the content of the request at that time. This information is critical when the end handler evaluate the request and decide to grant achievement to applicant. For example, to decide the opponent for a final thesis, the Dean want to know who the opponent of the final thesis proposed by the Head of Department is and why Guarantor disapproved with proposed opponent before the new opponent was proposed by the Guarantor.

Another important question is when an audit trail should be made to store this information of a request for auditing. At a phase, the handlers can edit instance at any time as they want until they transit the instance to the next phase. The system should allow user to edit the instance as they edit a document and save the current content of the instance. But saving an audit trail for auditing whenever the user saves the instance does not have much meaning at administration point of view. Handlers only care about the content of the instance before it leaves previous phases. Therefore, our system should only save an audit trail for auditing only right before the instance is transited to the next phase.

After storing this information, the records of an instance must be used to reconstruct the request at any time in its history. The reconstructed instance must show who is the editor of the content, at what time and what was the content. To store this information, we must design a proper data model, which will be described at the next section.

## Data model design

Show the whole relational data model (shortly data model) may be hard to observe and understand, therefore we break the data model into 4 parts, corresponds with 4 subsections. The whole data model is provided in the appendix. The last subsection provides an example of SQL query to demonstrate the function of designed data model.

The data model was normalized in Third Normal Form (3NF) as introduced by Codd E. F. in [17][18] and will only be introduced in 3NF in this section for concision. We also tried to simplify the data model as much as possible here to avoid rambling and ambiguity when solving a complex and broad problem. The data model in real implementation can be more complex to provide more functionalities, but we considered they does not affect the core problem, which we try to solve in this thesis, thus they will not be described in the following subsections.

To visualize the data model design, we used Entity Relationship Diagram (ERD) provided by opensource software StarUML. ERD in StarUML can be used to visualize a relational data model design. The drawback is, that it only shows the relationship as a connection line between two table without indication of which column of one table is referenced to the other. However, it provides signs, which can be used to show the reference between two table, those are the Primary Key (PK) and Foreign Key (FK). Other signs are also used in ERD are: Nullable (N), Unique (U) to indicate the correspond constraint in database. The ERD then can be used as a blueprint to create correspond collection of tables in the PostgreSQL with the help of SQLAlchemy. Hereafter we only use the term “table” instead of “entity” (used in ERD) or “relation” (used in relational data model), the term “column” instead of “attribute”.

The cardinality of a join between two table in ERD is presented as the figure below:

Text

Description automatically generated with low confidence

Figure 15 - ER Diagram cardinality

Source: StarUML

The data types of columns in the ERD, their correspond data types in PostgreSQL and their value are presented in the table below:

Table 1 - Data types of data model

| **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) |

Source: self-created

For naming convention, we used the plural noun with lower case for table name, because a table will contain many records. We used verbs for singular nouns in lower case to name one-to-one and one-to-many relationship because it is used to indicate the numerical relationship between one record in a table and one or more records on the other (the relationship many-to-many in relational data model is transformed to two one‑to‑many relationships with a bridge table and therefore it is not used in the following ERDs). We also used the lower case for naming attributes and the underscore (\_) for separating words.

### Users, groups, roles, positions

Diagram

Description automatically generated

Figure 16 - Data model of user, group, role, and position

Source: self-drawn

One indispensable table of our system is the users. Next, as described in section 4.1, other necessary tables of our system are groups, roles, and positions. Each entities have a minimum number of attributes e.g., id to identify each instance of entities, created\_at as a timestamp to indicate when the record is created, the name for each record, …

The users table will have a username and password attribute to store authentication information of each user.

The groups table have a relationship one-to-many with itself to model a hierarchy structure of the group in an organization.

In table roles, the column role is an enumerated type, which includes three possible roles, those are admin, handler, and applicant. These three roles are important for the authorization of our system. The user with role admin can create, update, delete groups, roles, positions and assigned position for other users. Moreover, the role admin is the only role can define an administration process and corresponding form. The role handler can either instantiate a form and initialize the process or can handle the instance in transit phase. The role applicant can only instantiate a form and initialize the process. The name of the roles is defined by the admin to reflect the actual roles of the organization e.g., Rector, Dean, Vice-Dean, Head of Department, …

The position, as discussed in 4.1, is derived from role and optionally from group. The group\_id as FK in positions table can be null as indicated in diagram above with character N and if the group\_id is null, the position and role are equivalent. This is helpful when we want to define a general position such as student, supervisor, etc., which do not belong to any group. In general, the purpose of the position is to be used to assigned for handling a phase and designated for redirecting instance to the next phase. Hereafter in the following text, the role admin, handler, or applicant are used to refer to the position derived from the correspond role.

The relationship between users and positions is many-to-many and is transformed to bridge table users\_positions.

### Administration process and form

The administration process is an entity of our system, and it can be a table in the data model. The administration process will contain multiple phases and as limited in the open of this chapter, is accompanied by only one form. This leads us to a question that can we use a form to represent an administration process? In this sense, there is no table processes, and the table forms will contain all the phases of an administration process (left diagram of Figure 17). We have tried to implement this idea and it is a simpler solution when only one form is defined for one administration process.

If multiple forms are defined for one process, we can add the table processes as being shown in the right diagram of the figure below. Each process can have multiple phases and multiple forms, and each form is used in several phases of the process. This solution can be considered as the unification of multiple subprocess into one process, where subprocess will have its own form with its own phases. This is an extended solution of the above one, because it only requires adding one table with two relationships. This allows us to implement the simple solution while keeping the ability for extending system in the future. In a limited time, we have only implemented the simpler solution and therefore in this section we only discussed about this simple solution.

Diagram

Description automatically generated

Figure 17 - Simpler (left) and extended (right) solution for data model of administration process

Source: self-drawn

As we introduced in subsection 3.1.3, the sequence of phases can be stored in two table phases and transitions. The relationship of these two tables can be understood, that the table phases has a relationship many‑to‑many with itself (One phase can have many next phases, and one next phase can have many phases) and therefore the relationship is transformed to a bridge table transitions with two one-to-many relationships.

Diagram

Description automatically generated

Figure 18 - Phases before and after transformation

Source: self-drawn

The structure of the form is created from sections and each section include fields as discuss in subsection 4.1.3. The sections and fields must be separate tables according to 1NF defined in [16][17] and have relationships as shown in left diagram of Figure 19. However, in each phase, some sections will be handled, and every phase of process belongs to the form. Therefore, we can use relationship one-to-many between phases and sections table (right diagram in Figure 19) instead of using relationship one-to-many between forms and sections. With this shift of relationship, we can know which form each section belongs to while we add the relationship between phases and sections.

Diagram

Description automatically generated

Figure 19 - Shifting relationship between forms and sections to phases and sections

Source: self-drawn

Positions are designated for phases and are assigned for sections of different process by the admin and the user with specified position will become a **potential director** and **potential handler** of these phases and sections. The complete data model of administration process is shown in the Figure 20.

Diagram

Description automatically generated

Figure 20 – Data model of administration process

Source: self-drawn

The phase\_type column of phases is an enumerated type, which includes 3 possible types of phases, those are begin, transit and end phase as introduced in subsection 4.1.1.

The order of phases, sections and fields is used for arranging their actual order in presentation of the form. The order is an integer from -32768 to +32767 (smallint in PostgreSQL) specified by the admin to decide the ascending order of phases, sections, and fields in the representation of the form and instance. All sections are arranged by the order of its containing phases in the form. Sections of each phase are arranged by the order of each section. Fields of each section are arranged by order of the field.

The public and obsolete of forms is used to indicate, whether the form is public or obsolete. Form is not public (private) by default when created by admin. Only private form can be read, updated, or deleted by the admin. The form, which is published by the admin, cannot be modified. User can only read and instantiated public form. Obsolete form can be read but cannot be instantiated by another user. The form is marked as obsolete only by the admin and only if the form is currently public. The instance of the obsolete form, which is not currently in the end phase can continue to be handled till it reach the end phase.

### Instances of a form

Diagram

Description automatically generated

Figure 21 - Data model of the instance

Source: self-drawn

As introduced in the subsection Choosing database, we split the structure of the form (sections and fields) from the data filled by user in each field of the form, which we call it the **content**. We must have a table, instances, to store the content. However, an instance has multiple contents for each field and therefore the contents must be split to a new table, contents, according to the 1NF [16][17]. The actual data filled by user is stored in column value of table contents. The column updated\_at is a time stamp indicated the last update of the value, which is critical for audit.

The administration process begins when a user instantiates an instance from the form and become the directors and the handler of the instance. A form can have multiple instances and an instance can have multiple directors and handlers, according to the phases and sections defined by the admin.

The instance can only appear only at one phase of the process at a time. This is true if the form is a sheet of paper, which cannot exist in two or more phase. At a particular phase the handler(s) of the phase can modify the assigned section(s) and then the director can redirect the instance to the next phase. Before director can redirect the instance, he must specify the handlers and director of the next phase as limited in 4.1.4.

At this point, there is one question, which we must consider: when can the director redirect the instance? While implementing the API, we have tried to introduce the states of the instance as it went through the phases to solve this question (the current\_state of instances and the resolved of the contents in the Figure 21). With these states, we can set a strict rule, when the director can redirect the instance. For example, the state “full resolved” indicates that all the handlers of one phase already handled their section(s) and mark their section as “resolved”. The director can only redirect the instance to the next phase only if the instance in the state “full resolved”. However, the problem happens when more scenarios was introduced to the system. For example, if the instance was sent to the next phase and handler of the next phase found some error in the instance, all the handlers of that phase must mark the instance as “resolved”, then the director is able to redirect the instance back to the phase, where the error occur. We can mitigate this problem by introducing more states if more scenarios are introduced. However, instead of trying to focus on every scenario, we use a very simple and loose rule, which can solve every scenario, that is: whenever the director redirects the instance to the next phase, the instance is at the next phase, regardless the instance is “resolved” or not. With this rule we also simplify the audit part of our system, which is described in the next subsection.

The content of a field in an instance must be created only once. We could have let the instance\_id and field\_id become the PKs of contents table and the unique constraint of PK will guarantee the single existence of a content for a field in an instance. However, the unique constraint can be created for these two fields explicitly when creating table or later altering the table (Figure 22 shows how to alter a table for adding unique constraint for multiple column). Then the column id is created as the PK of contents for identifying the content of the field in the instance. In this way we guarantee the unique constraint of the content of a field in an instance and provide a single reference of this content for the other table (another table is the table envelope, which we will introduce in the next subsection).

A screenshot of a computer

Description automatically generated with medium confidence

Figure 22 - Adding unique constraint for table contents explicitly in PostgreSQL

Source: self-made

Similarly, an instance at any phase only has one director and for any section in the instance, there is only one handler, and there for we create unique constraint on instance\_id and phase\_id for directors and another unique constraint on instance\_id and section\_id for handlers. The id of directors and handlers is used for uniquely identify the records of these table.

Text

Description automatically generated

Figure 23 - Creating table directors and handlers with unique constraint on multiple columns in PostgreSQL

Source: self-made

### Audit and audit trail

As introduced in section 1.3, audit trail is a chronological record of applicant’s request in each phase of administration process, that we can reconstruct and examine the sequence of phases, which the request traversed through in predefined administration process.

A chronological record of the applicant’s request mean that we must store the data about the request in the order of time. To reconstruct and examine the sequence of phases we must store the history of phases, which the instance traversed through. In each phase, we must retrieve a copy of the instance and check its whole contents, the handlers who lastly modified the contents and the last time they were modified. Only the moment right before the instance is transited to the next phase these data are stored and they are used as the audit trail. The action of creating an audit trail is called **committing** and audit trail is called a **commit**. The instance is auto committed after a director transit the instance to the next phase.

As introduced in subsection 3.5.1 we have seen that the approach of Git to store the commit history is similar to our requirement on the history of the instance through phases. To apply this approach in our system:

First, we must consider each content value in an instance is like a blob in Git, but instead hashing only the content (and the size of the file) of the blob like Git, we use a hash function to produce a hash from the id of the content, the id of the handler who lastly modified the content, the content value, and a timestamp of the last modification of the handler before the instance is transited to the next phase. The hash and all its parameters create what we call an **envelope**. Notice that instead of storing the instance id and the field id to uniquely identifies the content of a field in an instance, we only need to reference the envelope with a single content id, thus reduce the number of columns of the table. The content id also helps us to prevent two same content value of two different instances modified at the same time have a same hash. The creator\_id in another hand is the id of handler, who lastly edited the content. The receiver of the content can be changed by the director as discuss in subsection 4.1.4 and therefore the handler of the content must be presented to identified who lastly modified the content. The timestamp is needed to determine when the content is lastly modified before the instance is transited to the next phase. Two same content value edited by the same handler but in different time will become two envelopes with two different hashes. The correspond table will be envelopes with columns hash\_envelope, content\_value and instance\_field\_id.

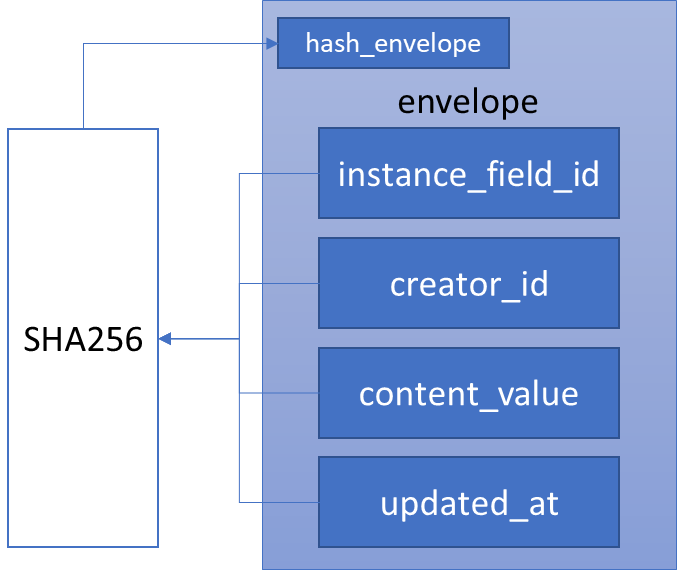


Figure 24 - Creating an envelope

Source: self-drawn

Note that we are not tightly bound to use any specific type of the hash function in this model. However, for later implementation of the API we chose to use SHA256 as the hash function as it is considered secure against collision in the time written this thesis [20][21]. Other hash functions can be an alternative if it is considered secure.

Next, we also need a tree as Git, but instead of using tree to store the name of the files and the hash of blob, or the name of folders (directories) and hash of tree, we only use tree to store the hash of the envelope. A **tree** is like a snapshot of the instance at a phase in process. The hash of the tree is created from all hash of its envelopes. The correspond table will have name trees with columns hash\_tree.

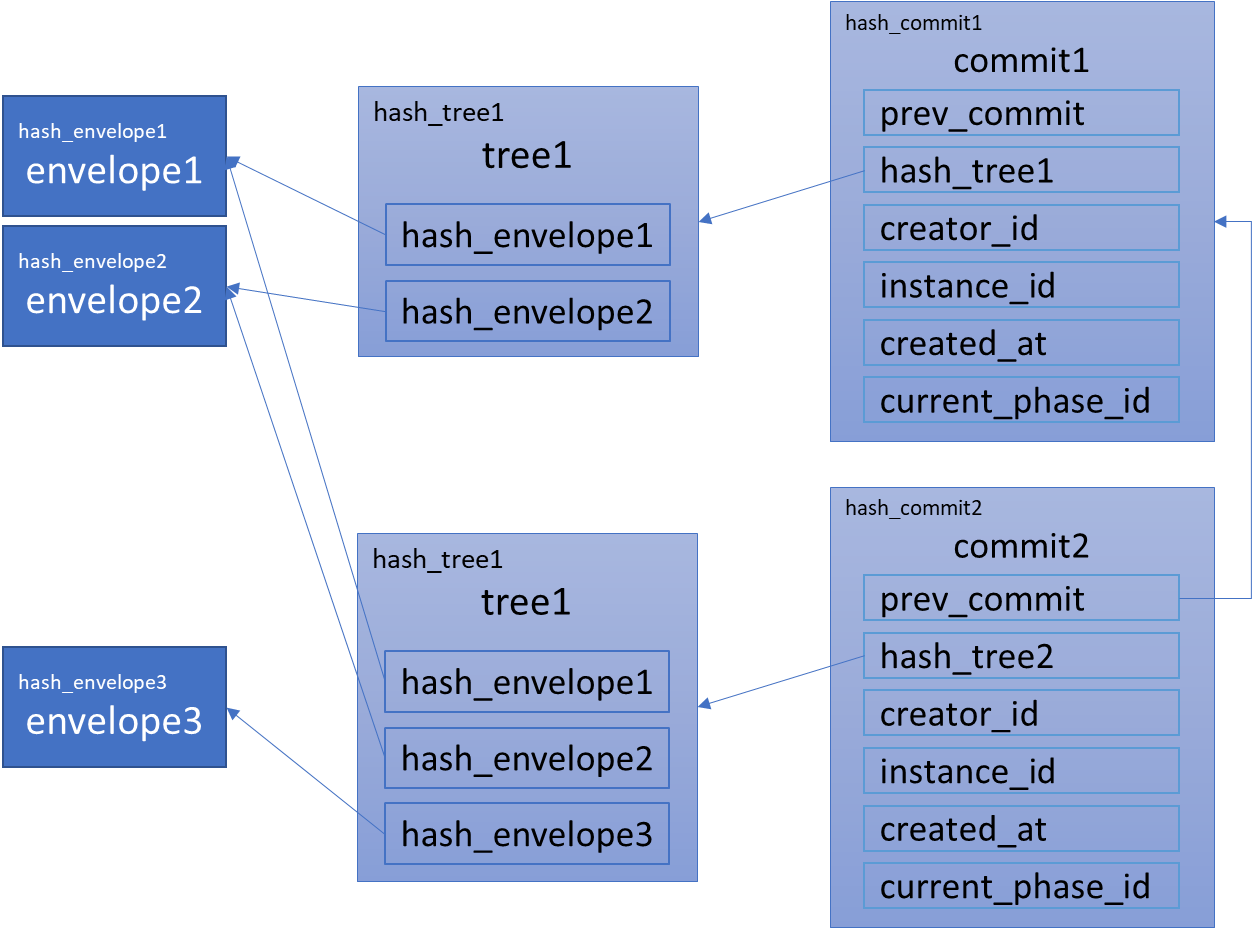


Figure 25 - An example of two commits

Source: self-drawn

Last, the commit will include the hash to previous commit if any, the hash of the tree, the creator id of the commit (the director id), the instance id, a timestamp when the commit is created, the phase id where the commit is created. These data are then used to create the hash of current commit. The table will have name commits with columns hash\_commit, prev\_commit, hash\_tree, creator\_id, instance\_id, created\_at, current\_phase\_id.

A complete data model for audit and audit trail is shown in the Figure 26. The hash value of the envelope, tree and commit are stored in the form of 64 hexadecimal digit as a string with the length of 64 characters in database.

Diagram

Description automatically generated

Figure 26 - Data model for auditing

Source: self-drawn

### Query examples

Queries, which we should consider are how to get a full instance; how to get the potential directors or potential receivers; how to get all directors and handlers of the instance; how to check if a user is director or receiver of the current instance; how to get the instance’s content at any commit in the commit history.

#### Get full instance

To get a full instance with its phases, sections, fields, and contents we must join 6 tables instances, forms, phases, sections, fields, instances\_fields. The example below shows a query to get full instance with a single required input, that is the instance id, and the output is the form name, phases name, sections name, fields name, and the contents value of corresponding fields order by the order of the phases, order of the sections and order of the fields. Other columns of these tables can be selected depending on the requirements of the query.

select \*

from instances join forms on instances.id = forms.id

join phases on forms.id = phases.form\_id

join sections on phases.id = sections.phase\_id

join fields on sections.id = fields.section\_id

left outer join instances\_fields on (

fields.id = instances\_fields.field\_id

and instances\_fields.instance\_id = instances.id)

or instances\_fields.id is null

where instances.id = 1

order by phases.order, sections.order, fields.order;

Note that in this query we use a left outer join between fields and instances\_fields to show all fields of the forms with or without content value. Figure below is an example result of the query, but we only select the name of form, phase, section, field, and the content value.

Graphical user interface, table

Description automatically generated

Figure 27 - The result of the query to get full instance

Source: self-made

With SQLAlchemy we can use the code as follows:

full\_instance\_1 = session.query(Instance, Form, Phase, Section, Field, InstanceField). \

join(Instance.form). \

join(Form.phases). \

join(Phase.sections). \

join(Section.fields). \

outerjoin(InstanceField, or\_(

and\_(InstanceField.field\_id == Field.id,

InstanceField.instance\_id == Instance.id),

InstanceField.id == None

)). \

filter(Instance.id == 1). \

order\_by(Phase.order, Section.order, Field.order).all()

The result is a list of tuples of ORM object of Instance, Form, Phase, Section, Field and optionally InstanceField. To prepare the result in form of a dictionary, reflecting the hierarchy between tables phases, sections, fields, instances\_fields, which is helpful for building a UI, we should loop through the result of the query as the code below:

instance = InstanceResponse.from\_orm(full\_instance\_1[0][0]).dict()

form = FormResponse.from\_orm(full\_instance\_1[0][1]).dict()

phases = {

'existed\_phases': dict(),

'phases': []

}

sections = {

'phase\_id\_key': dict(),

'section\_id\_key': dict()

}

for \_, \_, phase, section, field, instance\_field in full\_instance\_1:

instance\_field = InstanceFieldResponse.from\_orm(

instance\_field).dict() if instance\_field else None

field = FieldResponse.from\_orm(field).dict()

field['instance\_field'] = instance\_field

if section.id not in sections['section\_id\_key']:

section = SectionResponse.from\_orm(section).dict()

section['fields'] = [field]

sections['section\_id\_key'][section['id']] = section

if section['phase\_id'] not in sections['phase\_id\_key']:

sections['phase\_id\_key'][section['phase\_id']] = [section]

else:

sections['phase\_id\_key'][section['phase\_id']].append(section)

else:

sections['section\_id\_key'][section.id]['fields'].append(field)

if phase.id not in phases['existed\_phases']:

phases['existed\_phases'][phase.id] = phase.id

phase = PhaseResponse.from\_orm(phase).dict()

phase['sections'] = sections['phase\_id\_key'][phase['id']]

phases['phases'].append(phase)

full\_instance\_1 = {

'instance': instance,

'form': form,

'phases': phases

}

The result will be in form of a dictionary below. Note that instance and form is not added to the hierarchy.

{

'instance': {

# key:value for instance data

},

'form': {

# key:value for form data

},

'phases': [

{

# key:value for phase data

'sections': [

{

# key:value for section data

'fields': [

{

# key:value for field data

'instance\_field': {

# key:value for instance\_field data

}

}, ...

]

}, ...

]

}, ...

]

}

#### Get potential receivers, potential directors

To get the potential receivers we can join tables users, users\_positions, sections. The example below shows how to retrieve all potential receivers of section with id equal 1.

select users.\*

from users, users\_positions, sections

where users.id = users\_positions.user\_id

and users\_positions.position\_id = sections.position\_id

and sections.id = 1;

Similarly, to get all potential directors of the phase:

select users.\*

from users, users\_positions, phases

where users.id = users\_positions.user\_id

and users\_positions.position\_id = phases.position\_id

and phases.id = 1;

The result example of these queries is shown in the figure below:

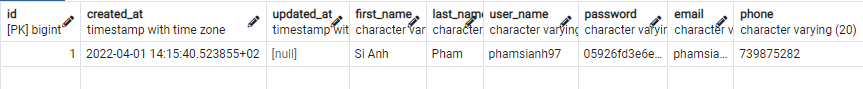


Figure 28 - The result of the query to get potential directors or potential receivers

Source: self-made

#### Get all receivers, directors

To get all receivers of the instance

select users.\*

from users, receivers

where users.id = receivers.user\_id

and receivers.instance\_id = 1;

Similarly, to get all director of the instance, we only replace the table receivers by table directors.

select users.\*

from users, directors

where users.id = directors.user\_id

and directors.instance\_id = 1;

The only input of these queries is the instance id. The output of these query is records of all users, who are directors or receivers of the instance.

#### Check if a user is director or receiver

To check if a user is a director of phase in an instance:

select true

from directors

where directors.user\_id = 1

and directors.phase\_id = 1

and directors.instance\_id = 1;

Similarly, to check if a user is a receiver of section in an instance:

select true

from receivers

where receivers.user\_id = 1

and receivers.section\_id = 1

and receivers.instance\_id = 1;

The inputs for this query are user id, instance id, and phase id. The output is True if the user is director of the phase in the instance and None if the user is not.

#### Get all content value of a commit

Suppose we know the hash commit (all commits of an instance can be query by joining instances and commits tables), first we must get all the envelopes of the commit:

select envelopes.\* from commits

join trees on commits.hash\_tree = trees.hash\_tree

join trees\_envelopes on trees.hash\_tree = trees\_envelopes.hash\_tree

join envelopes on envelopes.hash\_envelope = trees\_envelopes.hash\_envelope

where commits.hash\_commit = '97142a1be9b9ceecb83f33e5331354ca2719e695928b7084124ab660ec9420c9';

This query is used as a subquery in combination with the query in 4.2.5.1. The whole query is shown as follow:

select \*

from commits join instances on instances.id = commits.instance\_id

join forms on instances.form\_id = forms.id

join phases on forms.id = phases.form\_id

join sections on phases.id = sections.phase\_id

join fields on sections.id = fields.section\_id

left outer join instances\_fields on

(fields.id = instances\_fields.field\_id and instances\_fields.instance\_id = instances.id)

or instances\_fields.id is null

left outer join (

select envelopes.\* from commits

join trees on commits.hash\_tree = trees.hash\_tree

join trees\_envelopes on trees.hash\_tree = trees\_envelopes.hash\_tree

join envelopes on envelopes.hash\_envelope = trees\_envelopes.hash\_envelope

where commits.hash\_commit = '97142a1be9b9ceecb83f33e5331354ca2719e695928b7084124ab660ec9420c9'

) as envelopes on envelopes.instance\_field\_id = instances\_fields.id or envelopes.hash\_envelope is null

where commits.hash\_commit = '97142a1be9b9ceecb83f33e5331354ca2719e695928b7084124ab660ec9420c9'

order by phases.order, sections.order, fields.order;

If we only select the commit’s message, the instance id, the name of form, phase, section, field, the content value and the content value in the envelope from the result of the query above we can get the result as follows:

Table

Description automatically generated

Figure 29 - The result of the query to get full instance in a commit

Source: self-made

The code with SQLAlchemy to produce a similar query as above is:

hash\_commit = '97142a1be9b9ceecb83f33e5331354ca2719e695928b7084124ab660ec9420c9'

query\_envelopes = session.query(Envelope). \

join(Commit.tree). \

join(Tree.trees\_envelopes). \

join(TreeEnvelope.envelope). \

filter(Commit.hash\_commit == hash\_commit).subquery()

EnvelopeORM = aliased(Envelope, query\_envelopes)

full\_instance\_1 = session.query(Commit, Instance, Form, Phase, Section, Field, InstanceField, EnvelopeORM). \

join(Commit.instance). \

join(Instance.form). \

join(Form.phases). \

join(Phase.sections). \

join(Section.fields). \

outerjoin(InstanceField,

or\_(and\_(InstanceField.field\_id == Field.id, InstanceField.instance\_id == Instance.id),

InstanceField.id == None)). \

outerjoin(query\_envelopes,

or\_(query\_envelopes.c.instance\_field\_id == InstanceField.id, query\_envelopes.c.hash\_envelope == None)). \

filter(Commit.hash\_commit == hash\_commit). \

order\_by(Phase.order, Section.order, Field.order).all()

Similar to the Get full instance, we only need a single loop to get a dictionary that reflect the hierarchy of tables phases, sections, fields, instances\_fields, envelopes. We only modify the begin part of the loop (The later part is the same as in Get full instance) as follow:

for \_, \_, \_, phase, section, field, instance\_field, envelope in full\_instance\_1:

envelope = EnvelopeResponse.from\_orm(

envelope).dict() if envelope else None

instance\_field = InstanceFieldResponse.from\_orm(

instance\_field).dict() if instance\_field else None

field = FieldResponse.from\_orm(field).dict()

field['instance\_field'] = instance\_field

field['envelope'] = envelope

...

The envelope can be considered a copy of the content in the commit history; therefore, we place it in the same level as the content, that is right below the field.

## API design and implementation

### Creating API with FastAPI

To create a simple API in FastAPI with only one endpoint:

* Create an app instance from FastAPI class
* Create a function, which FastAPI calls the path operation function (POF)
* Decorate POF with a path operation decorator

The code in Python will be:

from fastapi import FastAPI

app = FastAPI()

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

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

    return {"form\_id": form\_id}

FastAPI automatically creates a file openapi.json, which is confront with the OpenAPI specification and from this file the interactive document will be created by Swagger UI as shown in the figure below:

Graphical user interface, application

Description automatically generated

Figure 30 - Interactive documentation created with Swagger UI

Source: self-made

### The pattern of resource, resource instance and related resource

When we experiment to create the API with FastAPI, we realize a pattern of resource, resource instance and related resource. According to Fielding: “Any information that can be named can be a resource” [5]. We apply this idea in our system, and we determine a resource, or we called it specifically a **resource instance**, will correspond to a record in each table. A **resource collection** is a collection of resource instance. A **related resource** can be a column of a table, a record of a table, a collection of records or any information derived from the data of one or many table, which are related to a resource instance e.g., the username of a user instance, the creator of a form, all fields of a form, all participants of an instance, …

Next, we use 4 HTTP methods in the conjunction with three types of resource, those are:

* GET method is used to retrieve a selected resource collection, resource instance or a related resource of a resource instance.
* POST method is used to create a new resource instance for a resource collection from the data enclosed in the request.
* PATCH method is used to update the resource instance attribute(s) correspond to the data provided in the request.
* DELETE method is used to remove a resource instance from a resource collection.

The result of combination of 3 types of resource and 4 HTTP method are 6 types of generic endpoint used in our API, which are summarized in the Table 2. The API consumer can make a request correspond to each endpoint.

Table 2 – 6 types of generic endpoint

|  |  |  |
| --- | --- | --- |
| HTTP method | URL path | Endpoint name |
| 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 |

### API architecture

In the creation API with FastAPI, the core logic for each endpoint will be implemented in the body of each POF. The logic includes authentication the user, check the constraint for each resource, validate data, process data with database, prepare the response, and send it. These logics are implemented through base class and its descendants, which I created or are provided by other libraries. A class diagram is shown in the Figure 31

SQLAlchemy provides a declarative base class derived from declarative\_base() function, which we refer in Figure 31 as BaseModel. The BaseModel is used for process data with database. For each table in our database, we will create a class inherited from this BaseModel class. The inherited class is called Object Relational Mapper (ORM). Each ORM is associated with a database table and each instance instantiated from an ORM is associated with a record in that table. The ORM can be used to create or delete table as DDL and is used mainly to insert, retrieve, update, delete data as DML, but with a higher-level abstraction. With ORM, we can process data in OOP paradigm in Python and without using SQL (although using SQLAlchemy ORM still requires the knowledge about SQL).

The pydantic library provides a BaseModel class, which we refer as BaseSchema[[2]](#footnote-2) class in Figure 31. It is used for data validation using python type annotation. The validation by pydantic is apply for every generic use case, but in our system, we use the BaseSchema only in 2 ways: validate user input in the request body and to filter the data in the response.

Diagram

Description automatically generated

Figure 31 - Class diagram for creating API with FastAPI

Source: self-drawn

The AuthorizationChecker is used for checking the role of the user in specific operation on a resource, for example: only admin can create an administration process and form, or for checking if the user has the ownership on a resource instance, for example: user can only change their own password. AuthorizationChecker use 4 schemas provided by BaseController, those are post\_schema, patch\_schema, delete\_schema, response\_schema (indicated as the association between these two class in Figure 31), for its checking authorization.

The BaseController is the base class for controlling the request, data processing and prepare the response. Class inherited from BaseController is called controller. Each controller will have its own model, which is the correspond ORM. Control the request mean 6 types of requests will be handle by 6 corresponds methods of the controller, which are used for generating POFs for FastAPI. 6 methods of BaseController implement the basic data processing and prepare the response without any specific constraint of the system. Controller inherited from BaseController will overwrite these 6 methods and implement its own data processing and prepare the response. Data processing mean that the system constraint on the resource must be check, only then an operation on data (e.g., insert, select, update, delete) is executed and committed to the database. The system constraint is specific for each problem domain, and in the case of our thesis, the system constraints were described throughout subsection 4.1 and 4.2. For example, an instance can only be created from a public form, or an instance can be transited to the next phase only if director is specified. Prepare the response mean data return from ORM must be checked, if it is a resource collection, a resource instance or a primitive data type and filter the data from database with schema before it returns the data as a response.

There are 4 types of schemas used for validating and filtering by BaseController and AuthorizationChecker. The post\_schema and patch\_schema is used for validating the input of user in the POST and PATCH request body, and for checking the role and/or the ownership of the user on inserting and updating the resource instance. The delete\_schema is used only for checking the role and/or ownership of a user on deleting the resource instance in a DELETE request. The response\_schema is used for filtering the data from database before the response is sent to the user.

The attribute related\_resources of controller is an array of string, where we can declare which are the related resource of a resource instance. This array must only contain string, which is the either public attribute, public method, or public property of correspond model class in model attribute. By default, the related\_resources will have all public attributes, public methods, and public properties of the correspond model class.

The Exception class is provided by Python. We create inherited classes from this class for our own custom exception. Each exception is raised when authentication, authorization, data validation or constraint of our problem domain is violated. Our custom exceptions are caught in each POF (indicated as the association between Exception and POFGenerator in Figure 31) and a HTTP response will be sent to user with an error status code.

BaseController is composited from BaseModel and BaseSchema and is aggregated with Exception. POFGenerator is composited from BaseController. It has 6 methods, which generate POFs for FastAPI from 6 methods of each controller. Each method of generator is where the authorization checker and controller interact with each other and is where the custom exceptions are caught. In addition, each generator method of POFGenerator will assign the docstring of the correspond controller method to provide the technical documentation of each endpoint of API.

The advantages of this design are the quick and easy creation and editing of an endpoint of a resource, the separation between data processing with database and system constraints, therefore it is easy to change the code in controller method when the system constraints change.

The disadvantage of this design are schemas must be created for each new table in the database, an endpoint must explicitly define if it does not follow the convention of 6 types of endpoints. However, when implementing the API, the most problems we have encountered is that the change of system constraint and the disadvantage of this design did not affect much when we create or change an endpoint.

## Technical documentation

The documentation for the API must provide any API consumer the knowledge on how to send a request to the API and get back the response, and how to construct a client application (the frontend or the user interface) from the API. To accomplish this assignment, we have divided the documentation into two parts. The first part we provide a low‑level documentation on how to send a request and get back the response. The second part we provide a high‑level documentation on how to combine the endpoints of the API to create a client application from this API.

### Low-level documentation

We accomplish the first part by using the autogenerated file openapi.json by FastAPI from the code implementing our API. This file can be used in the Swagger UI, which is an interactive documentation for sending request and get back the response. Any API consumer, who obtains a copy of the openapi.json file, can use it to generate an interactive documentation with Swagger UI or any autogenerated interactive documentation, which confronts with the standard OpenAPI.

In the low-level documentation we only focus on providing the information about: HTTP method, path parameter, query parameter, request schema, response schema and description for an endpoint of API. This information is enough for a consumer to create the request and get back the response on the endpoint. An example of a low-level documentation for an endpoint of API is shown in the figure below.

Graphical user interface, text, application

Description automatically generated

Figure 32 - An example of low-level documentation for an endpoint

Source: self-made

The openapi.json file is included in the appendix of this thesis.

### High-level documentation

The second part of documentation however requires more effort to accomplish. We must describe all the step that an API consumer can follow and construct their own client application. The high-level documentation is provided in the appendix of this thesis. In this documentation we only focus on important actions, which fulfill the requirements described in chapter 1 of this thesis, those are:

* How to define a sequence of phases (administration process)
* How to define who is/are responsible for every phase in administration process.
* How to define a structure of form.
* How to create an instance from defined form.
* How to transit instance through administration process.
* How to recover instance at any phase.

A minimalist user interface as an example for implementation of a client application is describe in the next subsection.

## Minimalist user interface

To demonstrate the functionality of the API, we use React to build a minimalist user interface with the help of library Cytoscape.js to visualize the phase and transition. The result is shown in the figures below.

Figure 33 shows how to create a phase for administration process in user interface. The phase creation includes specify the name, optionally description for the phase, phase type, designated position, and the order of the phase. The designated position is where we define responsible person to become the director of the phase. The phase type can either begin, transit or end as discussed in section 4.1. The order of the phase is used to arrange the order of sections, which belongs to this phase, in the form and instance.

Graphical user interface

Description automatically generated

Figure 33 - Creating phase in user interface

Source: self-made

The Figure 34 shows an instance created from a form with the name “APPLICATION FOR APPROVAL OF THE PROPOSAL TOPIC OF THE FINAL THESIS”. All information is filled by the student in the corresponding section and fields. Next to the right of the section name is the assigned position of the handler of the section. The black frame indicates a phase of the form. The information about the phase and its transition is provided in the context menu on the top right of the black frame.

Graphical user interface, text, application

Description automatically generated

Figure 34 - An instance example with the user interface

Source: self-made

Other actions on the instance are provided in the context menu on the top right of the instance, those actions are transiting the instance to the next phase, receiving a section of the instance, marking the instance as done, viewing the commits history, reading information about phases, transitions, directors, and receivers of the instance, changing the receiver of the instance. The Figure 35 shows the user interface when a director at the “Phase for Head of Department” is transiting the instance to the “Phase for Student”, where the director of the next phase and receiver of the section for student are already specified. The current director can specify the leading teacher for corresponding section of the next phase. The director can provide a message for the commit to express the meaning of the commit or additional information for all participants of the instance.

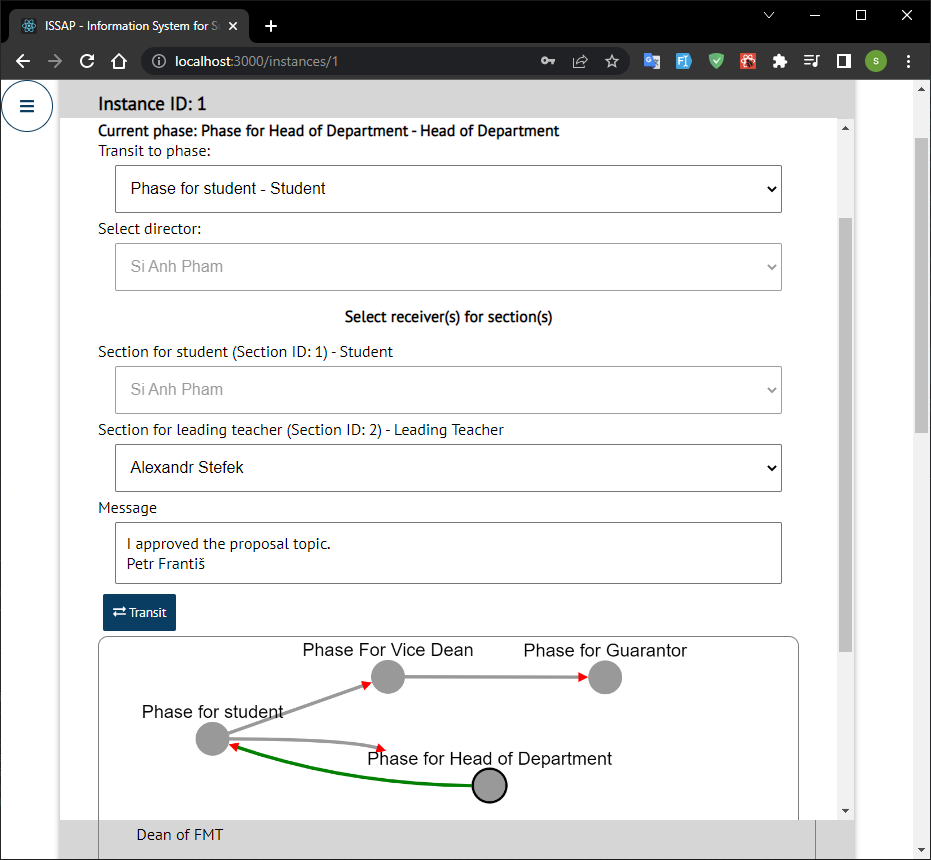


Figure 35 - Transiting the instance in the user interface

Source: self-made

The commit history of the instance is shown in the Figure 36. Each commit includes the commit’s created time, the phase where the commit is created, the first 16 hexadecimal digit of the hash, the name of the creator, and the message.

Table

Description automatically generated

Figure 36 - The commits history (audit trails) example of an instance

Source: self-made

## Containers with Docker Compose

When we use Docker Compose to create a set of containers, by default Compose sets up a bridge network, in which each container for a service joins the default network and is both reachable by other containers on that network, and discoverable by them at a hostname identical to the container name. The network created by Compose can be visualized in the Figure 37. We have 3 containers, those are:

* db: this is the container created from the official image postgres of PostgreSQL, which listens on port 5432.
* fastapi-app: the API container, which is served by Uvicorn[[3]](#footnote-3) and will listen on port 8000.
* react-app: the container for user interface, which is served by the Nginx[[4]](#footnote-4) on port 80. This is the only container that map to the port 80 of the host.

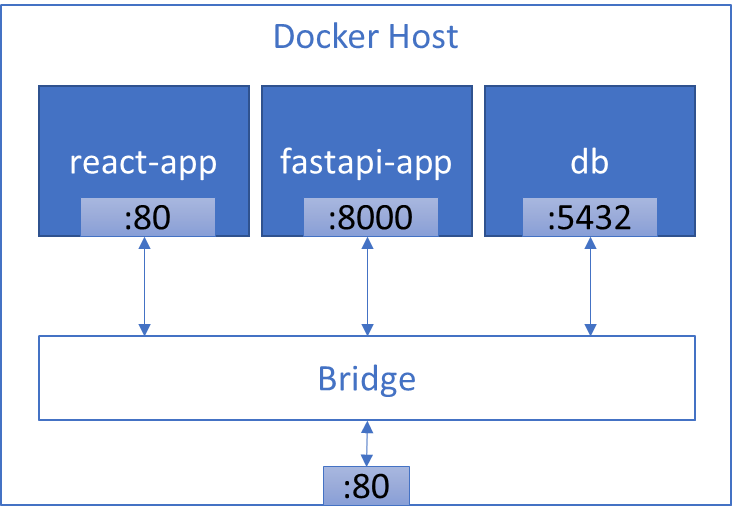


Figure 37 - Bridge network created by Compose

Source: self-drawn

After running the command npm run build, React will create a folder build, which includes static files e.g., a HTML file, a JavaScript file, a CSS file,… This folder can be served by Nginx as a static web page.

Graphical user interface, text, application

Description automatically generated

Figure 38 - An example of build folder created by React

Source: self-made

Moreover, Nginx can be used as reverse proxy to pass traffic of the API request of react-app to fastapi-app. A minimal configuration for Nginx to become a web server for static file and a reverse proxy will be:

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;

        }

    }

}

The Nginx server will listen at the port 80 as aforementioned. Traffic from the API request of the react-app will be passed to the fastapi-app at port 8000. The content of build folder will be put at the root directory /usr/share/nginx/html and serve as static content. Every request will be checked whether the specified file or directory in the URI exists. Otherwise, Nginx makes an internal redirect to index.html file at the root directory. (Note that we must include the MIME types to enable Nginx to serve static file like CSS, otherwise the header for these files will be Content‑Type: text/plain and the CSS will not be functional)

The communication between containers and the client is shown in the Figure 39. The client will make a HTTP request to the static index.html file served by Nginx. The index.html file will request all JavaScript and CSS file in the root folder. These 3 main files will create the user interface for the client. Depending on the action we made in the user interface, the corresponding API request will be sent to the Nginx and Nginx will pass the request to the fastapi-app. The fastapi-app will process the request, make queries to db, and check all the constraints. When the response is ready, fastapi-app will send it to Nginx and Nginx will pass the response to the client. The user interface will render the data for the user and the user will see the changes of the action they made in the user interface.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 39 - Communication between client and containers

Source: self-drawn

## The necessary output

The source code of the container providing the service to authenticated users, the documentation for the API, through which the service is available, and the source code of the container implementing the user interface are provided in the GitHub platform at the link <https://github.com/Phamsianh/diplomova_prace>. The structure of GitHub repository is divided into 3 directories: backend, documentation, frontend as the order of the required output.

The backend is directory, where we put the source code of the service (API) providing to the authenticated users. The Dockerfile in this directory is used to create the image for running the container.

The documentation includes the openapi.json file, which is used as a low-level documentation, and the documentation.docx as a high-level documentation. Other documentation and visualization can be found in this directory.

Similar to the backend directory, the frontend also includes a Dockerfile which is used for creating the image for running the container of user interface. The Dockerfile in this case is a multi-stage build, which we build our React user interface and Nginx as web server and reverse proxy.

Finally at the root of repository is the docker-compose.yml file, which is used with Docker Compose to create a set of containers for our system.

## Summary

In the first section we discussed about the administration process, in which the following constraints have been made: there is only one begin phase, one end phase and multiple transit phase; between two phases exists maximum two transitions; there is no transition from one phase to itself; each phase only have one director and possibly multiple handlers; A form structure includes sections, each section includes fields; Each section of an instance is handled by one handler at a time; director can specify or change the handler of the section; director of current phase of the instance can transmit the instance at any time and the system will audit the instance before it is transited to the next phase.

The next section described the data model including 18 tables, their relationships, and constraints of these tables. These tables are users, groups, roles, positions, users\_positions, forms, sections, fields, phases, transitions, instances, instances\_fields, directors, receivers, envelopes, trees, trees\_envelopes, commits.

Third section described the API implementation with FastAPI, which follows 6 types of generic endpoints, those are Get Resource Collection, Get Resource Instance, Get Related Resource, Post Resource Collection, Patch Resource Instance and Delete Resource Instance. We defined 3 classes and used with classes of other libraries to create the structure of the API, those are BaseController, AuthorizationChecker and POFGenerator, BaseModel by SQLAlchemy, BaseSchema by pydantic, APIRouter of FastAPI.

The technical documentation is divided into 2 parts, low‑level documentation, which describes how to create a request and get back the response of endpoints of API, and high-level documentation, which describes how to combine endpoints to use the functionalities of the system.

In the fourth section the user interface created with React was shown to demonstrate the functionalities of the API such as creating form, creating an instance, transiting the instance, and view the instance’s commits history.

The sixth section described 3 containers in form of a bridge network created by Docker Compose and communication between them. Those containers are: db, fastapi-app, react-app.

Finally, section 4.7 described the structure of the GitHub repository where we place all necessary outputs of the thesis.

# Conclusion

The goal of this thesis is to create an information system (with access via web) with functionality supporting electronic administration process with elements enabling an audit of the performed tasks. Three outputs are provided in the GitHub platform at the link <https://github.com/Phamsianh/diplomova_prace>, those outputs are:

* Source code of the container providing the service to authenticated users (backend directory).
* Documentation for the API through which the service is available (documentation directory).
* Source code of the container implementing the user interface (frontend directory).

The assignments of this thesis are:

* Design the information system as a set of cooperating containers (docker compose).
* Study API creation (according to the OpenAPI / Swagger standard) using the FastAPI library.
* Create an information system data model.
* Implement API.
* Prepare technical documentation for API.
* Create a minimalist user interface for the ability to demonstrate functionality.

To fulfill these assignments, we first researched about the term and technology included in the assignment, which is described in the first three chapters. Afterward we described the implementation of these assignments in the fourth chapter. The summary of each chapter is described as follows:

In the first chapter, we introduced about the goal of this thesis, requirements for our information system, assignments for achieving the goal, and the necessary output of the thesis. Administration process, audit and audit trail are also introduced to give the audience a brief overview of the problem domain, which were discussed and solved in fourth chapter.

In the second chapter, we researched about APIs in general, web-based application software and web APIs in particular, lead to an affirmation of what the information system with access via web is. We also researched about 3 types of web API: RESTful API, RPC and GraphQL. As a result, standard and technologies used in a RESTful API were used for creating our web API, those are URL, HTTP, JSON, HTML, JS, CSS.

Third chapter introduced technology stack, which was used in developing our information system, those technologies are: PostgreSQL as relational database management system, FastAPI as library for creating web API, React as framework for creating user interface, Docker as containerization technology, Git as version control system, and GitHub as source code hosting platform. The inspiration from Git for implementing audit was also introduced in this chapter.

Fourth chapter discussed about administration process in detail with consideration of requirements of the system. The implementation of thesis’s assignments was also described in the following order: The creation of information system data model is described in section 4.2; The study of API creation using FastAPI library and the API implementation is described in section 4.3; The preparation of technical documentation for API is described in section 4.4; The creation of a minimalist user interface for the ability to demonstrate functionality is described in section 4.5; The design of the information system as a set of cooperating containers with docker compose is described in 4.6. The necessary output is described in section 4.7.

In this thesis, we completely focused on achieving the goal of the thesis with fulfilling the requirements on the information system, followed the assignments and provided the necessary output. Sufficiently to say that we have accomplished the goal of the thesis, that is creating an information system (with access via web) with functionality supporting electronic administration process with elements enabling an audit of the performed tasks.

The benefit of this thesis is created information system for supporting electronic administration process, which allows users: to define a sequence of phases for administration process; to define responsible user for every phase; to define structure of form comprised of sections and fields; to instantiate an instance from defined form and transit it through phases; to create instance’s commits history and recover the instance at any phase in its history.

For future development, the following functionalities of the information system should be considered: the creation of an administration process comprised from multiple subprocess; the merger of processes into one process; the process between multiple organizations; the automatically specification of the handlers for sections of a phase depending on group of the user; the notification for the user when an instance is pending for receiving, handling, or transiting; specific data type for each field, … The legal effect of the signature or the seal of the handler in electronic form can be a challenging problem in implementing. REGULATION (EU) No 910/2014 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL [24], Article 25, point 2 stated about the legal effect of electronic signatures: *“A qualified electronic signature shall have the equivalent legal effect of a handwritten signature.”* and Article 35, point 2 stated about the legal effect of electronic seal: *“A qualified electronic seal shall enjoy the presumption of integrity of the data and of correctness of the origin of that data to which the qualified electronic seal is linked.”* To make the instance of a form have a legal effect in the future, the information system should head to integration with a qualified trust service provider, which provides qualified certificates to users in the system for creating qualified electronic signature or qualified electronic seal.

# References

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. Available from: doi:10.6028/nist.sp.800-53r5
4. CNSS 4009. *Committee on National Security Systems (CNSS) Glossary* [online]. Committee on National Security Systems. 2022. Available from: <https://www.cnss.gov/CNSS/issuances/Instructions.cfm>
5. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures* [online]*.* 2000. Available from: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
6. BLOCH, Joshua. *A Brief, Opinionated History of the API* [online]*.* 2018. Available from: <https://www.infoq.com/presentations/history-api/>
7. STAIR, Ralph M. and George W. REYNOLDS. *Fundamentals of Information Systems*. 6th ed. Boston, USA: Course Technology, Cengage Learning, 2012. ISBN 0840062184.
8. BERNERS-LEE, T., R. FIELDING and L. MASINTER. *Uniform Resource Identifier (URI): Generic Syntax* [online]. B.m.: RFC Editor. 2005. Available from: doi:10.17487/rfc3986
9. FIELDING, R., J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH and T. BERNERS-LEE. *Hypertext Transfer Protocol -- HTTP/1.1* [online]. B.m.: RFC Editor. 1999. Available from: doi:10.17487/rfc2616
10. FACEBOOK, INC. *GraphQL* [online]*.* 2021. Available from: <https://spec.graphql.org/October2021/>
11. OPENAPI INITIATIVE. *OpenAPI specification v3.1.0* [online]. 2021. Available from: <https://spec.openapis.org/oas/v3.1.0.html>
12. MOGODB, *Understanding the Different Types of NoSQL Databases* [online]*.* Available from: <https://www.mongodb.com/scale/types-of-nosql-databases>
13. DOCKER. *Use containers to Build, Share and Run your applications* [online]*.* Available from: <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]*.* Available from: <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. and B. STRAUB. *Pro Git* [online]. 2021. Available from: <https://git-scm.com/book/en/v2>
20. BLUECRYPT. *Cryptographic key length recommendation* [online]. Available from: <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. Available from: <https://nukib.cz/download/uredni-deska/Kryptograficke_prostredky_doporuceni_v1.0.pdf>
22. FIELDING, Roy Thomas. REST APIs must be hypertext-driven [online]. 2008. Available from: <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>
23. RICHARDSON L. and M. AMUNDSEN. *RESTful Web APIs*. Sebastopol, California: O’Reilly Media, Inc. 2013, ISBN: 9781449358068.
24. *Regulation (EU) No 910/2014 of the European parliament and of the council of 23 July 2014: on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC* [online]. THE EUROPEAN PARLIAMENT AND THE COUNCIL OF THE EUROPEAN UNION, 2014. Available from <https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1588426886500&uri=CELEX:32014R0910>
25. JIN, Brenda, Saurabh SAHNI and Amir SHEVAT. *Designing Web APIs.* Sebastopol, CA: O’Reilly Media, Inc. 2018. ISBN: 9781492026921.
26. Neo4j. *The Neo4j Cypher Manual v4.4*, Introduction [online]. Available from: <https://neo4j.com/docs/cypher-manual/current/introduction/>
27. IBM Cloud Education. *Rest APIs* [online]. Ibm.com, 2021. Available from: <https://www.ibm.com/cloud/learn/rest-apis>
28. RedHat. *What is a REST API?* [online]. Redhat.com, 2020. Available from: <https://www.redhat.com/en/topics/api/what-is-a-rest-api>
29. Microsoft. *RESTful web API design* [online]. Docs.microsoft.com, 2022. Available from: <https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design>
30. NALLY, Martin. REST vs. RPC: what problems are you trying to solve with your APIs? | Google Cloud Blog. *Google Cloud Blog* [online]. 2018. Available from: <https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-your-apis>
31. TIWARI, S. Professional NoSQL. Indiana: John Wiley & Sons, Inc., 2011. ISBN 9780470942246.

Appendices

[Appendix A 1](#_Toc104061728)

[Data model 1](#_Toc104061729)

[Appendix B 1](#_Toc104061730)

[High-level documentation for API 1](#_Toc104061731)

[Define a sequence of phases (administration process) 1](#_Toc104061732)

[Define responsible users for every phase in administration process 2](#_Toc104061733)

[Define a structure of form 3](#_Toc104061734)

[Create instance from defined form 4](#_Toc104061735)

[Transit instance through administration process 5](#_Toc104061736)

[How to recover instance at any phase. 6](#_Toc104061737)

Appendix A

Data model

Diagram, schematic

Description automatically generated

Appendix B

High-level documentation for API

A high-level documentation on how to combine endpoint to create a client application. A client application must provide the following functionalities:

* Define an administration process, which is a sequence of phase.
* Define responsible users for every phase in process.
* Define a structure of a form, which are sections and fields.
* Authenticated user can instantiate an instance from the form.
* Instance can be transit through phases.
* Reconstruct instance at any phase in the commit history.

These functionalities are break down in correspond sections below.

We use HTTP methods and the path in the URL to denote an endpoint, for example: POST /forms. Documentation on how to create a request and get back the response is provided in low-level documentation as file openapi.json. In this high-level documentation, we only describe how to combine endpoints to get the proper functionalities of the designed system.

Define a sequence of phases (administration process)

Steps for create an administration process:

1. Create form
2. Create phases
3. Create transitions

To create a sequence of phase, first we must create a form. We use the endpoint POST /forms to create a form. An example of the response body in JSON format is follow:

**{**

**"id": 1,**

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

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

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

**"public": false,**

**"obsolete": false**

**}**

In the response, we have the id of the new form, from which we can use to create the phases for this form.

To create a phase for the form, we use the endpoint POST /phases, in which we provide the id of the form in the field form\_id of request body. Each request returns the new phase’s information including the id of the phase, which we use to create transitions of process. An example of a response body is as follow:

**{**

**"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**

**}**

*Note: the creation of phase includes the definition responsible user for this phase. Please refer to the next subsection.*

To create a transition, we use the endpoint POST /transitions. In which we provide the id of the source phase in field from\_phase\_id, and the id of the target phase in field to\_phase\_id of the request body.

Best practice show that we should plan all phases and its transitions in the administration process and before creating it.

We can show all phases and transitions of the form through endpoint GET /forms/{form\_id}/phases and GET /forms/{form\_id}/transitions.

*Note: Only admin can create an administration process.*

Define responsible users for every phase in administration process

We know that for each phase there is only one director and for each section of this phase there only one handler. Responsible users are director and handlers. To define director and handlers for a phase, we use position to indicate that only user, who held the correspond position can become director or handlers of the phase. We say: a position is designated for a phase and a position is assigned for a section.

First, we must get all positions in the systems by using endpoint GET /positions. An example of a response body from this endpoint is as follow:

**[**

**...**

**{**

**"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**

**},**

**...**

**]**

A position id is used to provide in the field positions\_id of the request body of endpoint POST /phases to designate a position for this new phase. An example of request body for creating a phase is follow:

**{**

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

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

**"description": null,**

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

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

**"order": 1**

}

Similarly, we provide a position id in field position\_id of the request body of endpoint POST /sections to assign a position for this new section. An example of request body for creating a section is follow:

**{**

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

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

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

**"order": 1**

**}**

*Note: only admin can designate and assign positions for phases and sections of an administration process.*

Define a structure of form

A structure of a form includes sections and fields. Each phase of a form includes sections, and each section includes fields.

To create section, first we must know which phase the new section belongs to. The phase id then is used to provide as value of the field phase\_id in the request body of endpoint POST /sections. An example of the response body from this endpoint is as follow:

**{**

**"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**

**}**

*Note: the creation of section includes the definition responsible user for this section of a phase. Please refer to the previous subsection.*

To create a field, we use endpoint POST /fields, in which we provide section id as a value of the field section\_id to indicate that the new field belongs to this section. An example of the response from this endpoint is as follow:

**{**

**"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**

**}**

*Note: Only admin can define structure of the form.*

Create instance from defined form

To create an instance from a form, we use the endpoint POST /instances, in which we provide form id as a value of the field form\_id in the request body. Only user with position designated for the begin phase of the form can instantiate an instance. An example of a request body of this endpoint is as follow:

**{**

**"form\_id": 1**

**}**

An example of the response body is as follow:

**{**

**"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"**

**}**

When instance was created, the creator of the instance become the director of the begin phase and become the handler of all begin sections, which are assigned for creator’s position. All contents of these sections are auto created with the value of an empty string. This is necessary for creating the hash of the envelope.

To retrieve all directors and receivers of the instance, we use endpoint GET /instances/{instance\_id}/directors and GET /instances/{instance\_id}/receivers.

To retrieve all contents of the instance, we use endpoint GET /instances/{instance\_id}/instances\_fields.

*Note: Only participants of the instance can get related resource of instance resource.*

Transit instance through administration process

To transit (redirect) instance to the next phase, we use endpoint PATCH /instances/{instance\_id}. In the request body of this endpoint we must specified:

* The id of the next phase as a value of field current\_phase\_id.
* The id of the user, who will become a director of the next phase, as a value of field director\_id.
* The id of sections and id of the users, who will become handlers of sections of the next phase, as key/value pairs in the field receivers.
* The message for the commit, which auto created when the instance is transited to the next phase, as value in field message.

An example of request body is as follow:

**{**

**"transit": {**

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

**"receivers": {**

**"3": 5**

**},**

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

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

**}**

**}**

*Note: Only director of the current phase can transit (redirect) the instance to the next phase. Director must be specified but not all section’s handlers must be specified.*

User with designated positions is called phase’s potential director. User with position assigned for a section is called section’s potential receiver (or handler). We can get all potential directors of a phase by using endpoint GET phases/{phase\_id}/potential\_directors and can get all potential receivers of a section, using endpoint GET /sections/{section\_id}/potential\_receivers or GET /sections/{section\_id}/potential\_handlers.

The receiver can handle the instance by using the same endpoint as transiting the instance, PATCH /instances/{instance\_id}. The request body however is different. Receiver only mark the field handle to true. An example is shown below:

**{**

**"handle": true**

**}**

When instance was handled by the receiver, the receiver become handler of the sections and all content of these sections assigned for this receiver will be created. These contents can only be edited by its handler.

How to recover instance at any phase.

The commit is auto created by the system when the director redirects the instance to the next phase. The participants of the instance can retrieve all commits of the instance by using the endpoint GET /instances/{instance-id}/commits. An example of the history of commits is shown below, where contain only 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"**

**}**

**]**

To reconstruct the instance at any phase of the history of commits, first we must retrieve all current contents of the instance, then we must retrieve all envelopes of instance’s commit. The client application can use the content value of these envelopes with the structure of the form to inject to content of the corresponding fields. We use the end point GET  /instances/{instance\_id}/instances\_fields to get all the current contents of the instance and the endpoint GET /commits/{hash\_commit}/envelopes to retrieve all envelopes of the instance’s commit. An example of respond body for retrieving all contents of the instances is shown below:

**[**

**{**

**"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**

**},**

**...**

**]**

An example of respond body to retrieve all envelopes of a commit is shown below.

**[**

**{**

**"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. Other media type is introduced <https://www.iana.org/assignments/media-types/media-types.xhtml> [↑](#footnote-ref-1)
2. For clarity BaseModel for database, BaseSchema for validation [↑](#footnote-ref-2)
3. Uvicorn is a web server implementation for Python. [↑](#footnote-ref-3)
4. Nginx is an opensource web server that can also be used as a reverse proxy. [↑](#footnote-ref-4)