**UNIVERZITA OBRANY V BRNĚ**

**FAKULTA VOJENSKÝCH TECHNOLOGIÍ**

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

Studijní obor: Komunikační a Informační Technologie

|  |  |  |
| --- | --- | --- |
| Ev. číslo: |  | Počet listů: 66 |

**DIPLOMOVÁ PRÁCE**

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

Zpracoval: Bc. Pham Si Anh

Vedoucí práce: Prof. Dr. Ing. Alexandr Štefek

BRNO 2020

Copy of thesis assignmentsThanks pageAcknowledgement

Abstract

Keywords

Contents

[Abbreviations 9](#_Toc98511533)

[Figures 10](#_Toc98511534)

[Tables 12](#_Toc98511535)

[1 Introduction 13](#_Toc98511536)

[1.1 Goal and assignments for this thesis 13](#_Toc98511537)

[1.2 What is administration process? 14](#_Toc98511538)

[1.3 What is audit and audit trail? 15](#_Toc98511539)

[1.4 Thesis’s structure 16](#_Toc98511540)

[2 Web API 17](#_Toc98511541)

[2.1 Application Programming Interface (API) 17](#_Toc98511542)

[2.2 Web-based application software and Web API 18](#_Toc98511543)

[2.3 Information system with access via web 18](#_Toc98511544)

[2.4 REST and RESTful API 19](#_Toc98511545)

[2.4.1 REST 19](#_Toc98511546)

[2.4.2 RESTful API 19](#_Toc98511547)

[2.5 RPC 21](#_Toc98511548)

[2.6 GraphQL 21](#_Toc98511549)

[2.7 Other type of web API 22](#_Toc98511550)

[2.8 Summary 22](#_Toc98511551)

[3 Technology stack 24](#_Toc98511552)

[3.1 Types of databases 24](#_Toc98511553)

[3.1.1 Relational database 24](#_Toc98511554)

[3.1.2 Non-relational database 25](#_Toc98511555)

[3.1.3 Choosing database 28](#_Toc98511556)

[3.2 FastAPI 29](#_Toc98511557)

[3.2.1 FastAPI 29](#_Toc98511558)

[3.2.2 SQLAlchemy 30](#_Toc98511559)

[3.2.3 Swagger specification and OpenAPI specification 30](#_Toc98511560)

[3.3 React 31](#_Toc98511561)

[3.4 Docker 31](#_Toc98511562)

[3.4.1 Docker and container 31](#_Toc98511563)

[3.4.2 Networking in Docker and Docker Compose 32](#_Toc98511564)

[3.5 Git and GitHub 34](#_Toc98511565)

[3.5.1 The inspiration from Git 34](#_Toc98511566)

[3.6 Summary 36](#_Toc98511567)

[4 Problems to solve and implementation 37](#_Toc98511568)

[4.1 Administration process 37](#_Toc98511569)

[4.1.1 What is a phase in administration process? 37](#_Toc98511570)

[4.1.2 The designated position and redirector 40](#_Toc98511571)

[4.1.3 Structure of a form 41](#_Toc98511572)

[4.1.4 How an instance can be transited? 42](#_Toc98511573)

[4.2 Data model design 43](#_Toc98511574)

[4.2.1 Users, groups, roles, positions 45](#_Toc98511575)

[4.2.2 Administration process and form 47](#_Toc98511576)

[4.2.3 Instances of a form 51](#_Toc98511577)

[4.2.4 Audit and audit trail 53](#_Toc98511578)

[4.3 API design and implementation 56](#_Toc98511579)

[4.3.1 Creating API with FastAPI 56](#_Toc98511580)

[4.3.2 The pattern of resource, resource instance and related resource 57](#_Toc98511581)

[4.3.3 API architecture 59](#_Toc98511582)

[4.4 Technical documentation 62](#_Toc98511583)

[4.5 Frontend design and implementation 63](#_Toc98511584)

[4.6 Output 63](#_Toc98511585)

[Conclusion 64](#_Toc98511586)

[Source 65](#_Toc98511587)

# Abbreviations

|  |  |
| --- | --- |
| ISO | International Organization for Standardization |
| NIST | National Institute of Standards and Technology |
| CNSSI | Committee on National Security Systems Instruction |
| API | Application Programming Interface |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| HTTP | Hypertext Transfer Protocol |
| HTML | Hypertext Markup Language |
| IS | Information System |
| REST | REpresentational State Transfer |
| CRUD | Create Read Update Delete operations |
| JSON | JavaScript Object Notation |
| XML | eXtensible Markup Language |
| RPC | Remote Procedural Call |
| SQL | Structure Query Language |
| RDBMS | Relational Database Management System |
| ORM | Object Relational Mapper |
| CNM | Container Network Model |

# Figures

[Figure 1 - Example of administration process of an aplication for approval of the proposal of the topic of the final thesis 15](#_Toc98575805)

[Figure 2 - Relational database to store phases and transitions 28](#_Toc98575806)

[Figure 3 - The interactive documentation system auto created by Swagger UI from openapi.json 30](#_Toc98575807)

[Figure 4 - Containerization vs Virtualization. 32](#_Toc98575808)

[Figure 5 - Docker network design based on CNM 33](#_Toc98575809)

[Figure 6 - Visualization of blob, tree and commit 35](#_Toc98575810)

[Figure 7 - Visualization of a commit history 36](#_Toc98575811)

[Figure 8 - Illustration of a linear administration process. 38](#_Toc98575812)

[Figure 9 - Illustration of a non-linear administration process 40](#_Toc98575813)

[Figure 10 - Administration processes with redirectors 41](#_Toc98575814)

[Figure 11 - Form structure 42](#_Toc98575815)

[Figure 12 - ER Diagram cardinality 44](#_Toc98575816)

[Figure 13 - Data model of user, group, role, and position 46](#_Toc98575817)

[Figure 14 - Simpler (left) and extended (right) solution for data model of administration process 48](#_Toc98575818)

[Figure 15 - Phases before and after transformation 48](#_Toc98575819)

[Figure 16 - Shifting relationship between forms and sections to phases and sections 49](#_Toc98575820)

[Figure 17 - Complete data model of administration process 50](#_Toc98575821)

[Figure 18 - Data model of the instance 51](#_Toc98575822)

[Figure 19 - Adding unique constraint for table contents explicitly in PostgreSQL 53](#_Toc98575823)

[Figure 20 - Creating table redirectors and handlers with unique constraint on multiple columns in PostgreSQL 53](#_Toc98575824)

[Figure 21 - Creating an envelope 54](#_Toc98575825)

[Figure 22 - An example of two commits 55](#_Toc98575826)

[Figure 23 - Data model for auditing 56](#_Toc98575827)

[Figure 24 - Interactive documentation created with Swagger UI 57](#_Toc98575828)

[Figure 25 - Class diagram for creating API with FastAPI 60](#_Toc98575829)

# Tables

[Table 1 - Data types of data model 45](#_Toc98511613)

[Table 2 – 6 types of generic endpoint 58](#_Toc98511614)

# Introduction

## Goal and assignments for this thesis

Our goal 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. From this goal we must specify which functionality our system must support for administration process, and which performed tasks are audited. The requirements for the system on functionality supporting the 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 instance, 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.

Here 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 a request on the proposal of the topic of the final thesis to the guarantor of the study program (guarantor) through the Vice-Dean for study and pedagogical activities (VDSPA) with the form “Application for approval of the proposal of the 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. 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, the student will deliver the application to the Study Group of the Dean's Office of the Faculty.
* VDSPA will request the opinion of the guarantor on all Applications for student’s own final thesis topic. In case of non-approval of the proposed topic or other facts contained in the Application for its own final thesis topic, the guarantor is obliged to state specific reasons for non-approval.

We can visualize the process by Figure 1:

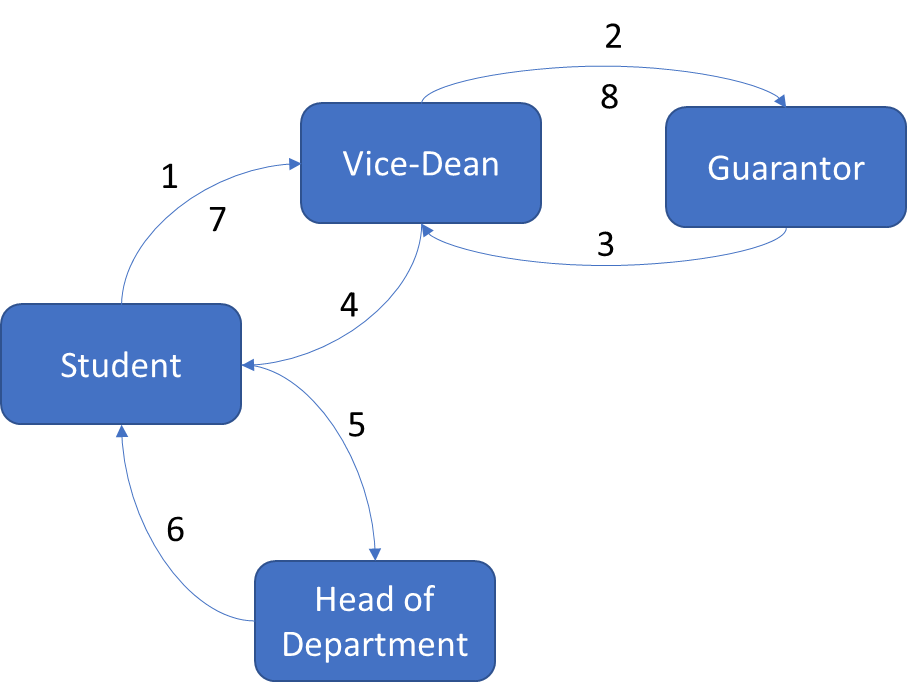


Figure 1 - Example of administration process of an aplication for approval of the proposal of the topic of the final thesis

Source: self-drawn

In general, the process will be defined by a regulation and often with a form. Applicant will initialize 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 doesn’t 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 determined 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 discuss the advantages and disadvantages of 3 types of web API. Then we explain why we choose RESTful API for this thesis.

In chapter 3 we provide an overview about technology stack, which are used for development our information system.

In chapter 4 we discuss in more detail about administration process and auditing on performed task, what is the problem we face in development, how we solve it and implement our information system.

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 look like, how to feed the input to the computer program and how the output should look like. 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 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 call 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 write 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 solved end-user’s need for a specific task, which can be solved without applying this program. Writing a document can be solved with pen and paper, but it can be solved with word processor. Accounting and auditing can be solved with pen and paper, but it 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 try 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, … we do not try to solve in this thesis, because it’s not essential part of the problem.

Our IS must be accessed via web as defined in [1.1](#_What_is_administration), 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 type or style of designing a web API.

## 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 describes 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.

REST has been developed to represent the model for how the modern Web should work. As a result, the rationale of REST is used to guild the standards or technologies in web e.g., URI, 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] . The term URL is a more specific term in context of this thesis; thus, we will use 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, …

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 between API consumer and with almost every type of resource. The exception is the interaction of a resource with another resource(s), 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 maintainer or consumer.

A data format of representation of a resource in REST API is often, but not always, JSON or XML. In this thesis we use 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] , 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.

Pros:

* Easy to implement, maintain and 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 when we want to retrieve representation of multiple resources.

## RPC

Remote Procedural Call (RPC) In distributed systems is when a client program calls a function whose implementation is in a remote server. In Web API, RPC-style APIs focus on actions. Each endpoint represents an action the client can perform on the server. The action is a function that is called on the server. Like a normal function that 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 is places in the URL and the arguments are placed either in the query string or body.

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

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. 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.

GraphQL reduce the redundant data in payload of a response and number of HTTP roundtrips in the comparison with REST API. It 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.

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 do not use GraphQL in this thesis, but it is considered a potential solution when our 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 web API include SOAP API or event-driven web API e.g., Webhook, WebSocket, HTTP Streaming, which are either no longer in use or server-to-server communication or bidirectional client-server real-time communication or sending an infinite chunk of data in infinite numbers of response, are considered unnecessary for the requirement of our IS, so we won’t make any further discussion about them 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, we discussed about what is API, web API, IS with access via web and three types of web API, which are mostly used in web development. We evaluate 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 will 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 requirements of the IS, the possibility to implement, 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 we think Docker 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 is an entity in our system, and it contains a varying number of fields in it. These fields are the structure of the form, and each form will have its own structure. Our system must be able to store such administration process and such arbitrary structures of different forms. Users must be able to initialize an administration process and instantiate an instance from corresponding form and fill data to it. Our system must be able to store current phase of the instance and data filled by user. We will consider two types of modern databases, which can fulfill these requirements, 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 invented 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. The schema is predefined and is not often changed, because change in schema required redefining database and data migration. Therefore, relational database is not ideal for unstructured data, and it slow down the development process because database schema usually changes overtime. However, with normalization, which is introduced by Codd in his first and second paper [16] [17] , we can reduce redundancy, the need for redefining database schema (restructuring), undesirable (anomaly) operation while keeping consistency on data we store.

The most common way of interacting with relational database systems is using Structured Query Language (SQL), which allows for specific queries to be built that help build, search, and filter data across one or multiple tables.

A relational database management system (RDBMS) is an application software, which provide an API and possibly a user interface and allows us to create, maintain, modify, and manipulate a relational database.

The most popular RDBMS are Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite, MariaDB, …

### Non-relational database

A non-relational database, sometimes called NoSQL (Not Only SQL), is any kind of database that doesn’t use the tables, fields, and columns structured data concept from relational databases.

There are four types of NoSQL Databases [12] :

#### Document Databases

Document databases store data in documents, which are usually JSON-like structures that support a variety of data types. 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, which usually store documents that have similar contents. Not all documents in a collection are required to have the same fields, because document databases have a flexible schema.

Retrieving data from document database vary between document databases and often is 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 database as:

SELECT \* FROM users

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

#### Key-Value Stores

Key-value databases (also known as key-value store) stores data in a “key-value” format and optimized for reading and writing data. The data is identified by a unique key or several unique keys. 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 have no query language. Values cannot be queried or searched upon. Only the key can be queried. Data is written (inserted, updated, and deleted) and queried based on the key.

Redis is the typical examples of key-value store.

#### Column-Oriented Databases

Column-Oriented databases store data by column rather than by row as relational database.

In relational database each row contains 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 (instance’s attribute). The problem happens if we want to retrieve a column or a small groups column of all rows, because 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 examples of column-oriented databases are Apache Cassandra, HBase.

#### Graph Databases

Graph databases solve the limitation of relational databases in schema by treating relation as data. The data of entity instance is stored in nodes and the relation 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 relation 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, use Cypher Query Language to retrieve data from graph. It was inspired by SQL and has similarity with SQL.

### 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 in the same process, and the transition is created for store that relation. 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 relations between entities are prone to change. Phase and transition are only two entities in our system and the relationship between them are 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 relation between these two table in chapter [4](#_Problems_to_solve).

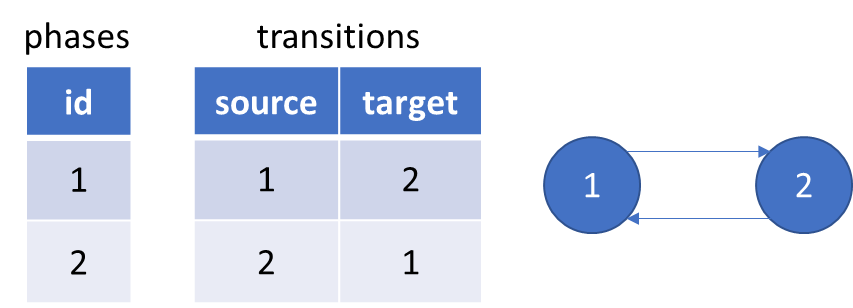


Figure 2 - Relational database to store phases and transitions

Source: self-drawn

The form is the structure for the 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, we must store this data in a manner, that we can retrieve this data 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 in documents with JSON-like format. 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.

But also, as being said we can 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. We will discuss about this solution in the chapter [4](#_Problems_to_solve).

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 time limitation in implementing our information system, I as the author of this thesis decided to choose relational database to develop our IS. The RDBMS I choose to use in this thesis is PostgreSQL.

## 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 API 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 3 - The interactive documentation system auto created by Swagger UI from openapi.json

Source: self-made

### SQLAlchemy

The language used to implement web API is Python. Python most well-known database drivers for PostgreSQL is psycopg2. But to simplify the development process we use SQLAlchemy, a Python SQL toolkit and Object Relational Mapper (ORM).

### 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 ReactJS 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.

## Docker

The first assignment of this thesis is to design the information system as a set of cooperating containers (docker compose).

### 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 4 - 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 container 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] .

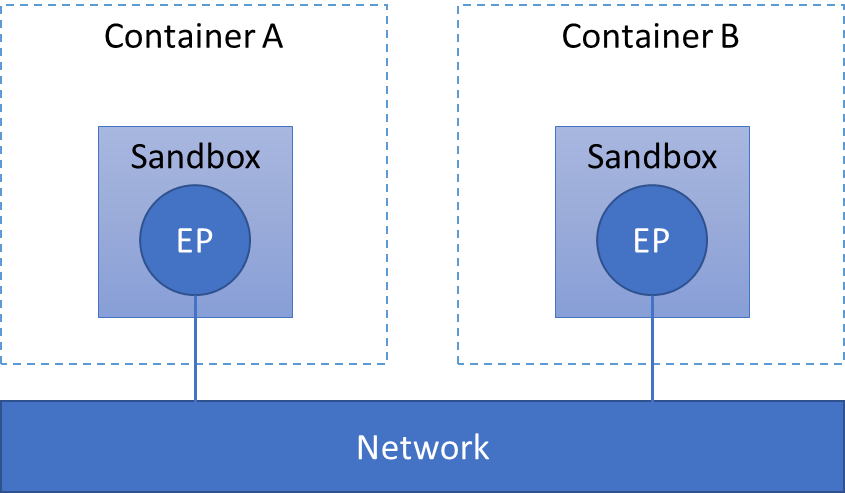


Figure 5 - Docker network design based on CNM

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. A user-defined bridges provide automatic DNS resolution between containers.

Docker Compose (shortly Compose) is a tool for defining and running multi-container Docker applications. By default, Compose sets up a single network for our app. 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).

Docker can also allow 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.

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 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.

As a necessary output of this thesis, we used Git and GitHub to manage our source code. The source code of our system is available on <https://github.com/Phamsianh/diplomova_prace>. Beyond this purpose 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 filename and the hash of a group of blobs, or the directory name and the hash of another tree(s). The tree object is like a director in UNIX file system and the blobs 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 doesn’t 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 6 is an example of the blob, tree, commit from the book *Pro Git* written by Scott Chacon, Ben Straub.

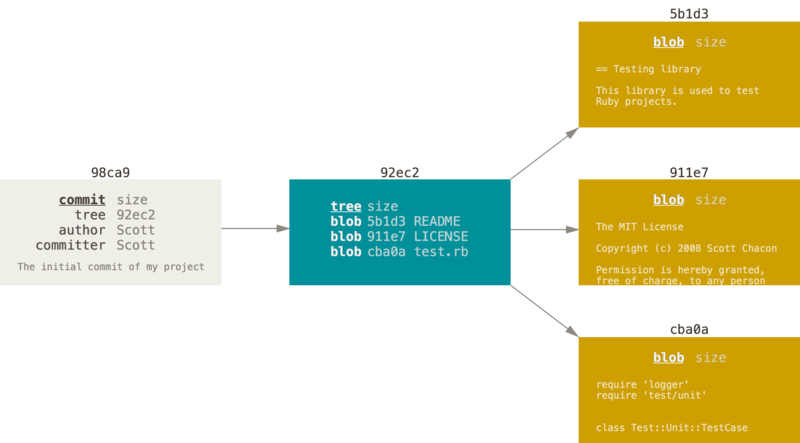


Figure 6 - Visualization of blob, tree and commit

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. This approach requires more computation than Git approach. 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 is 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.

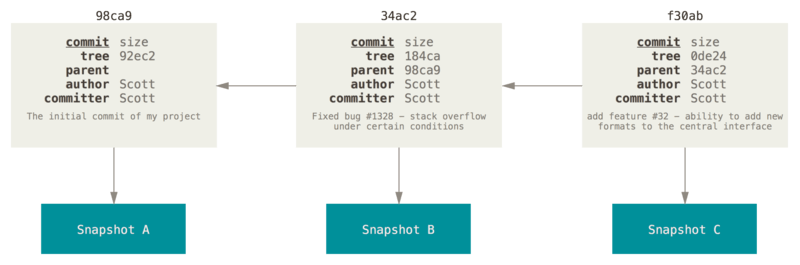


Figure 7 - Visualization of a commit history

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 section 4.2.4.

## Summary

In this chapter, we introduced 2 types of databases, those are relational and non-relational database and I as the author of this thesis decided to choose PostgreSQL as database management system to store data. We also introduced other framework and library to develop our IS, those are FastAPI, React and Docker. These framework and library are chosen in consideration with the requirements on our IS. The next chapter will present the design of our IS and how we implement our IS using these framework and library.

# Problems to solve and implementation

## Administration process

In an organization an administration process is defined by a regulation as a sequence of phases and is usually accompanied by 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... 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 doesn’t 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.

### What is a phase in administration process?

A **phase** is separate part of a process, assigned for position(s) to handle. The person handling the instance in a phase is the handler of that phase and 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 real life there can be only one begin phase, only one end phase and multiple transit phases. 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.

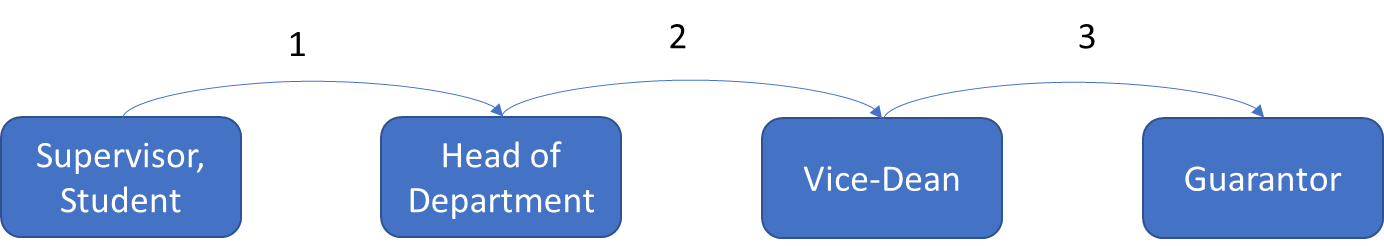


Figure 8 - Illustration of a linear administration process.

The numbers indicate the order of transition.

Source: self-drawn

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 the comment of the HoD. 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.

The above example only shows a linear process, where an instance is transit to the next phase without the possibility to turn back. As a result, it does not describe all the possible transitions in real life. For example, if HoD finds a typo of student, there is no transition back to the first phase. However, the transition is not described didn’t 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 doesn’t 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.

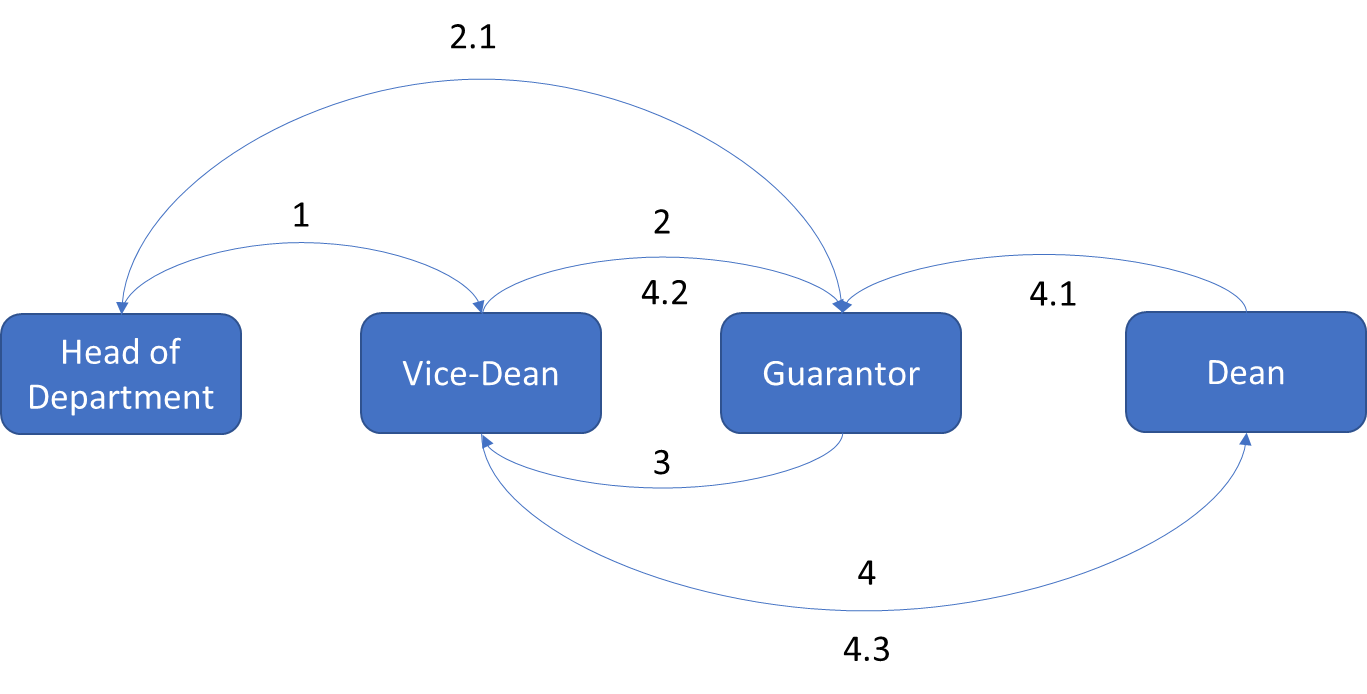


Figure 9 - Illustration of a non-linear administration process

Source: self-drawn

### The designated position and redirector

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 **redirector**, because the redirector redirects the instance to the next phase in the process. For example, the redirector of the begin phase in first example is the student and the redirectors of all other phases are the handlers of those phases.

The designated position isn’t necessary one of the assigned positions of a phase and the redirector in such case only performs the redirection, not instance modification. Note both above examples, the VDSPA doesn’t 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 redirector and not the handler of these phases (Figure 10).

Graphical user interface

Description automatically generated

Figure 10 - Administration processes with redirectors

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 can be handled, and each section can contain multiple fields. In real life each field in a valid instance is filled by only person with assigned position. The structure of a form can be visualized in the figure below.

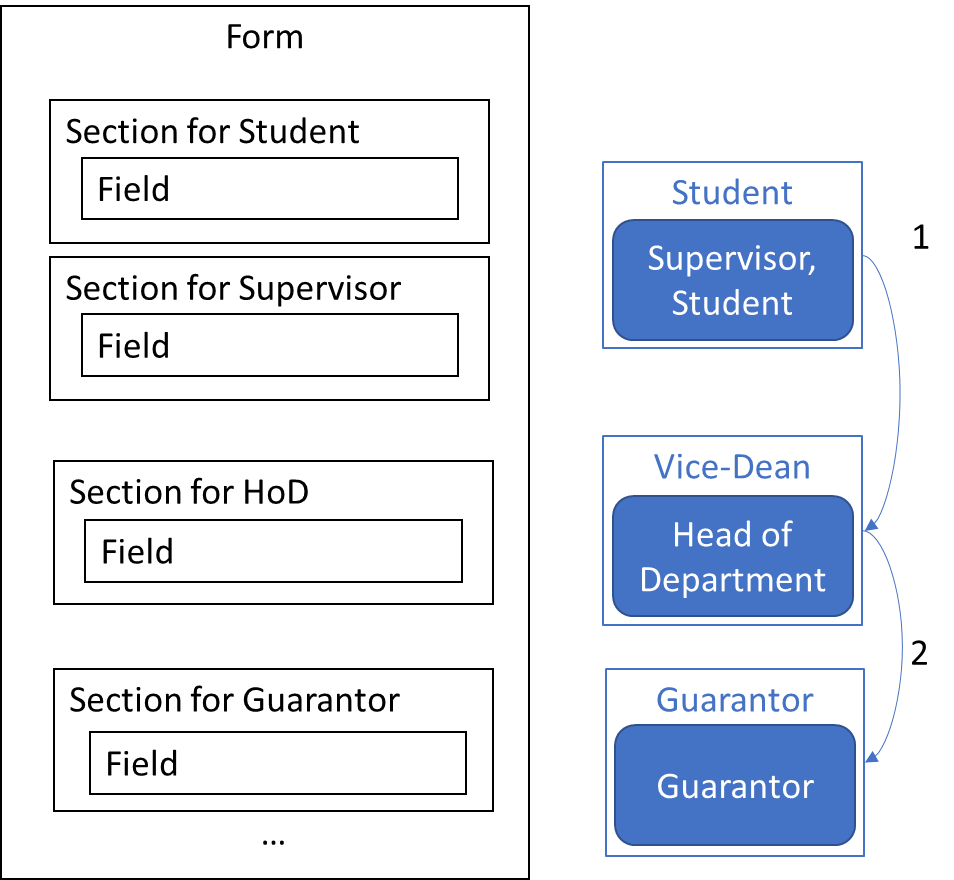


Figure 11 - Form structure

Source: self-drawn

### How an instance can be transited?

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 doesn’t belong to the organization but the 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, doesn’t 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 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. When one request handled by a handler, the next request is allocated for this handler.

For simplicity of this thesis, we only consider the first case, that is the applicant, and more generally the handler of each phase, must know who the handler(s) of the next phase are. The second case will be considered for future development.

After the instance is received, the handler will modify the fields assigned for him/her and the redirector transit the instance to the next phase. The process will continue till the end phase.

## 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 3 parts. In each part, we will discuss in detail.

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 trying to solve a complex and broad problem. The data model in real implementation can be more complex to provide more functionalities, but we considered they doesn’t 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 12 - 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. The underscore (\_) is used to separate words.

### Users, groups, roles, positions

Diagram

Description automatically generated

Figure 13 - Data model of user, group, role, and position

Source: self-drawn

One indispensable table of our system is the users. The users table in 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 don’t 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 user and position 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 14). We 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 is a form with its own phases. This is an extended solution of the above one, because it's required 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 only implemented the simpler solution and therefore in this section we only discussed about this simple solution.

Diagram

Description automatically generated

Figure 14 - Simpler (left) and extended (right) solution for data model of administration process

Source: self-drawn

As we introduced in section [3.1.3](#_Choosing_database), 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 15 - 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](#_Structure_of_a). 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 14. 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 14) 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 16 - 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 be a **potential redirector** and **potential handler** of these phases and sections. The complete data model of administration process is shown in the Figure 17.

Diagram

Description automatically generated

Figure 17 - Complete data model of administration process

Source: self-drawn

The phase\_type column of table 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](#_What_is_a).

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 18 - 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 redirectors and the handler of the instance. A form can have multiple instances and an instance can have multiple redirectors and handlers, according to the phases and sections defined by the admin.

The instance can only appear only at one phase of the process. 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 redirector can redirect the instance to the next phase. Before redirector can redirect the instance, he must specify the handlers and redirector of the next phase as limited in 4.1.4.

At this point, there is one question, which we must consider: when can the redirector redirect the instance? While implementing the API, we tried to introduce the states of the instance as it went through the phases to solve this question (the current\_state of instance and the resolved of the contents in the Figure 18). With these states, we can set a strict rule, when the redirector 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 redirector can only redirect the instance to the next phase only if the instance in the state “full resolved”. However, the problem happens when we there is 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 redirector is able to redirect the instance back to the phase, where the error occur. We can mitigate this problem by introducing more state and transition for more scenarios. 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 redirector 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 simply 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 (and only by one handler). 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 create table or later alter the table (Figure 19 show 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 (other 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 19 - Adding unique constraint for table contents explicitly in PostgreSQL

Source: self-made

Similarly, an instance at any phase only has one redirector 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 redirectors and another unique constraint on instance\_id and section\_id for handlers. The id of redirectors and handlers is used for uniquely identify the records of these table.

Text

Description automatically generated

Figure 20 - Creating table redirectors 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 modified the contents and the time it was modified right the moment before the instance is transited to the next phase. Only the last state of the instance before it is transited to the next phase is used to audit. We consider that the information about each modification of the handler doesn’t have much meaning in auditing the administration process. The action of creating an audit trail is called commit. The instance is auto committed after a redirector transit the instance to the next phase.

As introduced in subsection 3.5.1 we see 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 content value, the content id, and a timestamp of the last modification of the handler before the instance is transited to the next phase. The content id uniquely identifies the content of a field in an instance with its handler, and therefore we don’t need to store information about the handler. 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 hash, the content value, and the content id create an envelope. The correspond table will be envelopes with columns hash\_envelope, content\_value and instance\_field\_id.

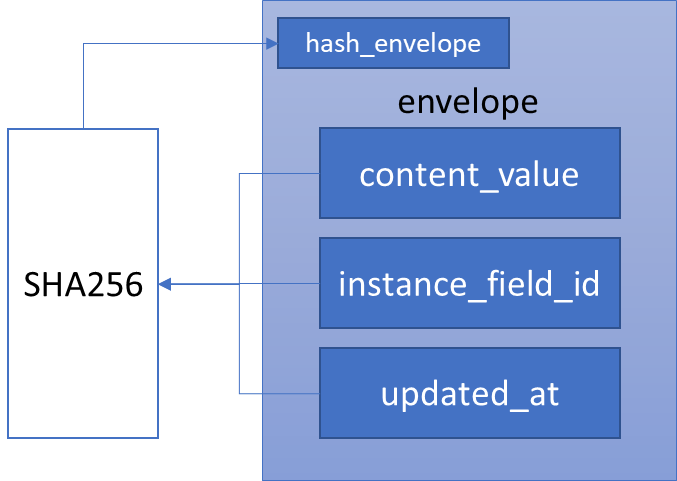


Figure 21 - Creating an envelope

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 or folders (directories) and the hash of blob or 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.

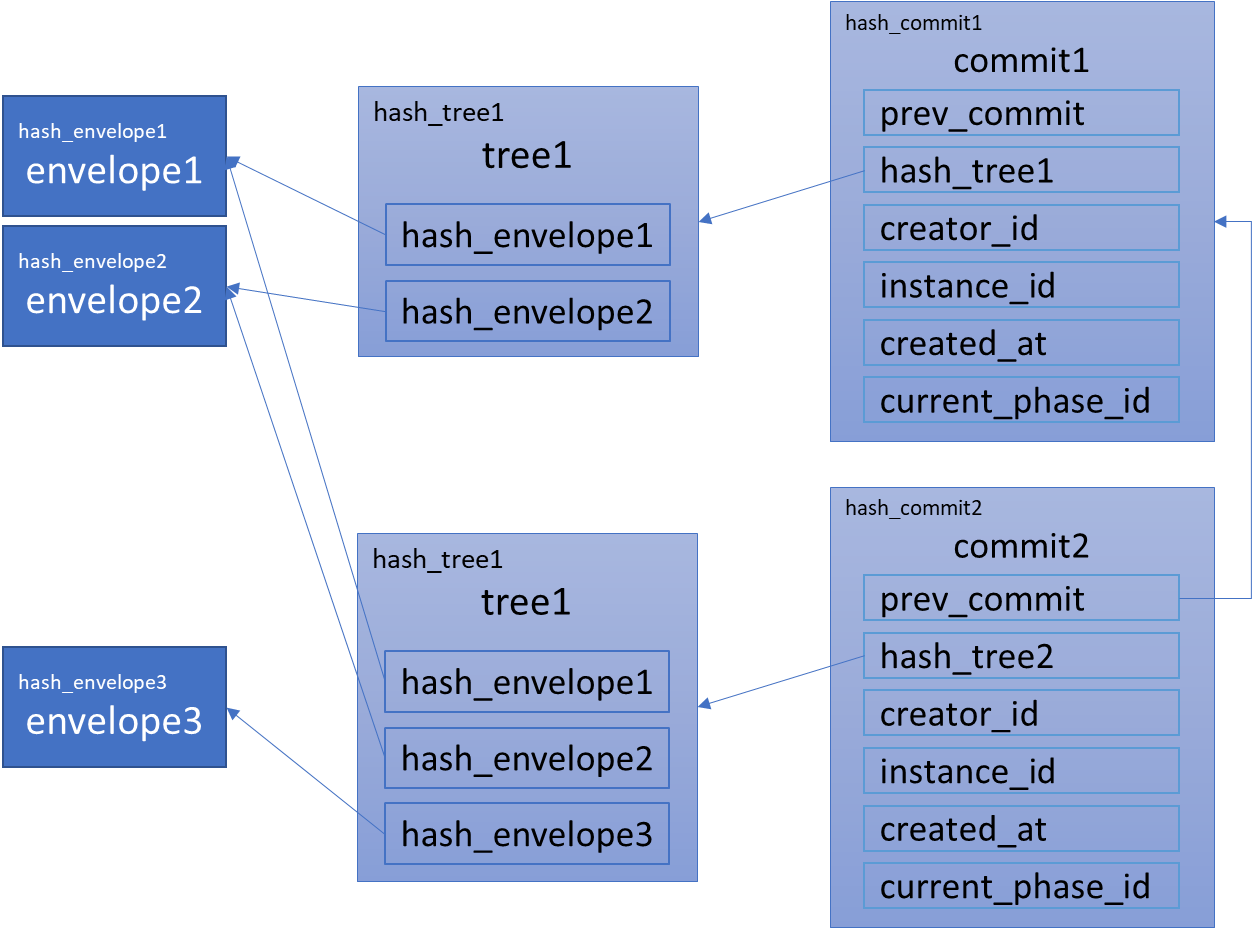


Figure 22 - An example of two commits

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 redirector 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 23.

Diagram

Description automatically generated

Figure 23 - Data model for auditing

## API design and implementation

### Creating API with FastAPI

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

* We must create an app instance from FastAPI class
* Create a function, which FastAPI call 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 24 - 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, there are only 4 HTTP methods, which is used in the conjunction with three types of resource in this thesis (other HTTP methods PUT, HEAD, CONNECT, OPTIONS, TRACE are not used), 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 25

SQLAlchemy provides a declarative base class derived from declarative\_base() function, which we refer in Figure 25 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[[1]](#footnote-1) class in Figure 25. It’s 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 25 - Class diagram for creating API with FastAPI

Source: self-drawn

The AuthroizationChecker 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 25), 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 redirector and all handlers are 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 model class. 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 25) 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’s 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’s doesn’t follow the convention of 6 types of endpoints. However, when implementing the API, the most problems we encountered is that the change of system constraint and the disadvantage of this design didn’t 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 divide 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. 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 26 - 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.

## Frontend design and implementation

## Output

# Conclusion

# Source

1. Hayes R., Dassen R., Schilder A., Philip W., *PRINCIPLES OF AUDITING, An Introduction to International Standards on Auditing.* Second edition. 2005. ISBN 0 273 68410 8
2. International Organization for Standardization, *ISO 19011*, *Guidelines for auditing management systems.* Third edition. 2018.
3. NIST *Security and Privacy Controls for Information Systems and Organizations.* <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf>
4. CNSS 4009. *Committee on National Security Systems (CNSS) Glossary.* <https://www.cnss.gov/CNSS/openDoc.cfm?X9siqniuE4GU0To+emCGBA==>
5. Fielding R.T., *Architectural Styles and the Design of Network-based Software Architectures.* 2000. <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
6. Bloch J., *A Brief, Opinionated History of the API.* 2018. <https://www.infoq.com/presentations/history-api/>
7. Stair R. M., Reynolds G. W., Fundamentals of Information Systems. 6th edition. 2012. Course Technology. ISBN-13: 978-0-8400-6218-5
8. Berners-Lee T., et al., *Uniform Resource Identifier (URI): Generic Syntax.* 2005. <https://datatracker.ietf.org/doc/html/rfc3986>
9. Fielding R., et al., *Hypertext Transfer Protocol -- HTTP/1.1.* 1999. <https://datatracker.ietf.org/doc/html/rfc2616>
10. Facebook, Inc., *GraphQL.* 2015. <https://spec.graphql.org/July2015/>
11. GitHub repository of OpenAPI specification <https://github.com/OAI/OpenAPI-Specification/>
12. MogoDB, *Understanding the Different Types of NoSQL Databases.* <https://www.mongodb.com/scale/types-of-nosql-databases>
13. Docker, *Use containers to Build, Share and Run your applications.* <https://www.docker.com/resources/what-container>
14. Poulton N., *Docker Deep Dive*, *Zero to Docker in a single book. 2020 edition.*
15. React, *Reconciliation.* <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. STUDIJNÍ ROZKAZ DĚKANA FAKULTY VOJENSKÝCH TECHNOLOGIÍ, Zadávání a zpracování závěrečných prací v AR 2021/22
19. Chacon S., Straub B., Pro Git, 2021, <https://git-scm.com/book/en/v2>
20. BLUECRYPT. Cryptographic key length recommendation. <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 [cit. 2020-04-18]. <https://nukib.cz/download/uredni-deska/Kryptograficke_prostredky_doporuceni_v1.0.pdf>
22. <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>
23. Richardson L., Amundsen M., *RESTful Web APIs*, 2013, ISBN: 978-1-449-35806-8

# Appendix

## 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 section 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 functionality 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 redirector and for each section of this phase there only one handler. Responsible users are redirector and handlers. To define redirector and handlers for a phase, we use position to indicate that only user, who held the correspond position can become redirector 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.*

### How to create an 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 in 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 redirector of the begin phase and become the handler of all sections, which are assigned for creator’s position. All contents of these sections are auto created.

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

### How to 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 redirector 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": {**

**"2": 6,**

**"3": 6**

**},**

**"director\_id": 6,**

**"message": "The proposal topic of thesis."**

**}**

**}**

*Note: Only redirector of the current phase can transit (redirect) the instance to the next phase. All sections handlers and redirector must be specified.*

User with designated positions is called phase’s potential redirector. 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 redirector 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":"a56d46d3319ae7e137ed3d5fd510b893ac3b969f66b35db39be692b71a499fff",**

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

**"hash\_tree": "9bdee3c7766be5ef82219df36c87cffd3865da3a9ff21b0006e87e61350a863f",**

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

**"instance\_id": 3,**

**"created\_at": "2022-03-21T20:26:04.220824+01:00",**

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

**"message": "The proposal topic of thesis."**

**}**

**]**

To reconstruct the instance at any phase of the commit, we must retrieve all the contents of that phase. The client application can use these contents with the structure of the form to inject the content into the corresponding fields. We use the endpoint GET /commits/{hash\_commit}/envelopes to retrieve all envelopes of the instance at a particular phase. An example of respond body is shown below, where there is only one envelope.

**[**

**{**

**"hash\_envelope": "a7f1fdef32510458e8976b4041892be5f195f810e2ff4d8fc985811107b6740c",**

**"instance\_field\_id": "6",**

**"content\_value": "Thesis created by Pham Si Anh",**

**"updated\_at": "2022-03-21T22:15:58.921408+01:00"**

**}**

**]**

1. For clarity BaseModel for database, BaseSchema for validation [↑](#footnote-ref-1)