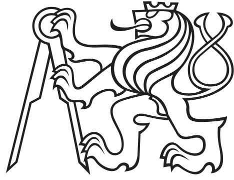
Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Graph database services extension

Bc. Filip Uhlík

Supervisor: RNDr. Marko Genyk-Berezovskyj
Field of study: Open Informatics
Subfield: Cybersecurity
May 2022

Acknowledgements

I would like to thank my supervisor, RNDr. Marko Genyk-Berezovskyj, for all the good advice he has provided me througout the writing process. I would also like to thank Ing. Ondřej Votava, for his patience and help with the integration of the application with the CTU single sign-on service.

Declaration

I hereby confirm on my honor that I personally prepared the present academic work and carried out myself the activities directly involved with it. I also confirm that I have used no resources other than those declared. All formulations and concepts adopted literally or in their essential content from printed, unprinted or Internet sources have been cited according to the rules for academic work and identified by means of footnotes or other precise indications of source. The support provided during the work, including significant assistance from my supervisor has been indicated in full.

In Prague, 20. May 2022

Abstract

The thesis documents the analysis, design and implementation of an extension to the Web Graph Service. The extension revolves around providing unified access to graph collections found on the Internet, by importing them into the application's database. The code and some problematic parts of the original application as well as the deployment process are analyzed in the first part of the thesis.

The extension is designed as a backend service written in Python. The new service oversees the contents of the graph database and provides a management API only accessible to administrators. The design part focuses on choosing a suitable web framework for the new service as well as improving some of the problems found during analysis. Automation of the deployment process is also designed using GitLab CI/CD pipelines.

The implementation part of the thesis describes some of the implementation details with the help of code snippets of the new application and the pipeline configuration.

The result of this work is an extended application that can import collections from various Internet sources. It supports three different file formats in which collections can occur and three different graph formats that the collections typically use. The application is available at `http://graphs.felk.cvut.cz`.

Keywords: graphs, graph collections, web application, python, backend, database, docker, ci/cd, gitlab

Supervisor: RNDr. Marko Genyk-Berezovskyj

Abstrakt

Práce dokumentuje analýzu, návrh a implementaci rozšíření Webové Grafové Služby. Rozšíření má za úkol poskytnout jednotný přístup ke sbírkám grafů dostupných na Internetu jejich importem do databáze aplikace. V první části práce je analyzován kód, některé problematické části a proces nasazení původní aplikace.

Rozšíření je navrženo jako backendová služba napsaná v Pythonu. Nová služba dohlíží na obsah databáze grafů a poskytuje API pro její správu, přístupné pouze administrátorům. Návrhová část se zaměřuje na výběr vhodného webového frameworku pro novou službu a také na vylepšení některých problémů zjištěných při analýze. Automatizace procesu nasazení je také navržena pomocí GitLab CI/CD pipeline.

Implementační část práce popisuje některé detaily implementace pomocí ukázek kódu nové aplikace a konfigurace pipeline.

Výsledkem práce je rozšířená aplikace, která dokáže importovat kolekce z různých Internetových zdrojů. Podporuje tři různé formáty souborů, ve kterých se mohou kolekce vyskytovat a tři různé grafové formáty, které kolekce typicky využívají. Aplikace je dostupná na adrese `http://graphs.felk.cvut.cz`.

Klíčová slova: grafy, grafové kolekce, webové aplikace, python, backend, database, docker, ci/cd, gitlab

Překlad názvu: Rozšíření služeb grafové databáze

Contents

Project Specification 1

1 Introduction 3
1.1 Feature overview . . . 3
1.1.1 Database contents . . . 3
1.1.2 Adding graphs . . . 4
1.1.3 Retreiving graphs from the database . . . 4
1.1.4 Graph collections . . . 4
1.1.5 Deployment . . . 4

2 Analysis 5
2.1 Requirements . . . 5
2.1.1 Graph collections . . . 5
2.1.2 Importing collections . . . 6
2.1.3 Graph formats . . . 6
2.2 Original application structure . . . 9
2.2.1 node-www . . . 9
2.2.2 node-www/compute . . . 10
2.2.3 db_update . . . 11
2.2.4 db_counter . . . 12
2.3 Deployment . . . 13
2.3.1 Dockerized applications . . . 14
2.3.2 CI/CD . . . 15

3 Design 17
3.1 Importing collections . . . 17
3.1.1 Administrator access . . . 19
3.2 Extended application structure . 20
3.3 Database schema . . . 20
3.4 Communication with the database 21
3.4.1 SQLAlchemy . . . 22
3.5 Database migration . . . 22
3.6 Web framework choice . . . 23
3.6.1 Django . . . 23
3.6.2 Flask . . . 24
3.6.3 Connexion . . . 26
3.6.4 FastAPI . . . 27
3.6.5 Result . . . 28
3.7 API design . . . 29
3.7.1 POST /compute . . . 29
3.7.2 GET /collection . . . 29
3.7.3 POST /collection . . . 30
3.7.4 GET /collection/{name}/{resolution} . 30
3.7.5 DELETE /collection/{name} 31
3.8 Manage collections UI . . . 31
3.9 Deployment . . . 32

4 Implementation 35
4.1 Application structure . . . 35
4.1.1 Config . . . 36
4.1.2 Database . . . 36
4.1.3 Exception . . . 37
4.1.4 Model . . . 38
4.1.5 Routers . . . 39
4.1.6 Services . . . 39
4.1.7 Utils . . . 45
4.2 Dockerization . . . 47
4.3 Docker Compose . . . 49
4.4 GitLab Pipeline . . . 49

5 Conclusion 53
5.1 Future improvements . . . 53

Bibliography 55

Appendix 58

A Graph formats 58

B Source code 59

Figures

2.1 House of Graphs collection example 7
2.2 Brandon McKay data page 8
2.3 Encyclopedia of Graphs collection download 8
2.4 Compute & insert page 11
2.5 Complete collections page 13

3.1 Collection lifecycle swimlane diagram 18
3.2 Collection table diagram 21
3.3 Model-Template-View diagram . 24
3.4 Flask route example 25
3.5 OpenAPI endpoint definition . . . 26
3.6 Python type hints example 27
3.7 Manage page wireframe 32
3.8 Deployment diagram 34

4.1 Configuration class example 36
4.2 Main Config class 37
4.3 Database query encapsulation . . 37
4.4 Exception handler example 38
4.5 Graph properties enum 39
4.6 Router function definition 40
4.7 FastAPI dependency injection . . 40
4.8 Compute properties function . . . 42
4.9 Download graphs function 44
4.10 Convert functions 45
4.11 Basic functions from the *collection* service 46
4.12 Calling functions annotated with *@with_session* 46
4.13 Dockerfile implementation 48
4.14 Docker compose service example 49
4.15 Service variables template 50
4.16 Production variables template . 51
4.17 Job created by a combination of templates 51
4.18 Build and deploy jobs templates 52

Tables

3.1 Web framework features 29

4.1 Pipeline jobs matrix 50

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Uhlík Filip** Personal ID number: **474489**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Cyber Security**

II. Master’s thesis details

Master’s thesis title in English:

Graph database services extension

Master’s thesis title in Czech:

Rozší ení služeb grafové databáze

Guidelines:

Propose and implement an extension of application http://graphs.felk.cvut.cz/.
The extended application will offer unified access to graph data files located in Internet outside the FEE CTU domain. The access will be structured analogously to the access to the internal data of the current application.
Propose the architecture of the extension and realize it. In particular, concentrate on the following:
-- Mechanism of integrating data collections obtained from Internet in the extended application and a unified form of data presentation to a user.
-- Automating a process of deployment of current and future modifications of the entire application.
-- Administrator and user roles in the extended application.
Demonstrate the operation of the extended application using graph data obtained from the given repositories. Optionally, include also other repositories according to your choice.
https://hog.grinvin.org/
http://atlas.gregas.eu/
http://users.cecs.anu.edu.au/~bdm/data/
Describe the development instruments used in the extended application and provide programmer‘s documentation of the whole project.

Bibliography / sources:

[1] Jonathan L. Gross, Jay Yellen: Graph Theory and Its Applications, Chapman and Hall/CRC, 2018
[2] Robert Sedgewick: Algorithms in C++ Part 5: Graph Algorithms, : Addison-Wesley Professional, 2002
[3] Martin Mareš, Tomáš Valla: Pr vodce labyrintem algoritm , CZ.NIC, z.s.p.o., 2017

Name and workplace of master’s thesis supervisor:

RNDr. Marko Genyk-Berezovskyj Department of Cybernetics FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **11.02.2022** Deadline for master's thesis submission: ___________

Assignment valid until: **30.09.2023**

RNDr. Marko Genyk-Berezovskyj
Supervisor’s signature

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Dean’s signature

 © VUT v Praze, Design: VUT v Praze, VIC

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Chapter 1
Introduction

Web Graph Service, or Graphs in short, is a web application hosted at `http://graphs.felk.cvut.cz/`. It has been created and extended by other CTU students in the past. Part of the application is a database with a considerable amount of graphs and their computed properties (such as the maximum and minimum degree of a vertex, whether the graph is asymmetric, biconnected, hamiltonian, a tree etc., 41 properties in total). Only undirected, simple, non-weighted graphs are supported. The application UI then allows the user to retrieve the graphs according to selected properties. The goal of this work is to add support for importing graphs from graph collections found in various Internet sources into the database, calculate their properties and give the user the option to filter and view the imported graphs.

1.1 Feature overview

The application has been deployed in production since 2018 and has been providing a set of features, which are described in this section. This section also gives a brief overview of the changes made to these features as a result of this work. The version of the application from the year 2018 will be throughout the thesis referred to as the original application. The version of the application that is the result of this work will be referred to as the extended application.

1.1.1 Database contents

The database of the original application contains all graphs up to the order of 10 vertices. Two properties, *genus* and *toroidal*, were not computed for almost any of the graphs larger than 3 vertices. The database of the extended application currently contains all the graphs up to 9 vertices, while the graphs on 10 vertices are still being recomputed. In addition to that, it also contains around a 1000 larger graphs up to the order of 150 vertices, that were added during the development. The extended application attempts to compute all the properties within a time limit, including the two properties ignored by the original application.

1.1.2 Adding graphs

When adding graphs to the original application, they need to be copied to the server and a script must be executed to compute the selected properties. The extended application can add single graphs via a part of the web UI, which does not work in the original application, or multiple graphs by importing a collection of graphs from the Internet. Even though the collection must be stored on a specific URL, the users do not have to rely on external sources. They can create their own collections and put them for example on their personal webpage, or any other URL. Graph collections are introduced in the Section 2.1.1.

1.1.3 Retreiving graphs from the database

The original application allows the users to view the graphs stored in the database via a web user interface. The UI allows to filter the graphs based on their properties. This feature is preserved in the extended application. The filtering functionality has been extended by the option to limit the search to a context of one or more graph collections.

1.1.4 Graph collections

The original application provides a static information about the complete collections of graphs present in the database. This whole feature is removed from the extended application in favor of the new interactive feature that provides the users with the option to suggest their own collections for import and interactively search graphs stored in these collections.

1.1.5 Deployment

The original application is deployed manually and runs directly on the server machine with a large amount of undocumented external dependencies, that need to be installed on the system. The extended application is dockerized into multiple docker images with all the dependencies preinstalled. The docker images are built and deployed on the server in an automated CI/CD pipeline.

Chapter 2

Analysis

The chapter discusses the application extension requirements in more detail. It also describes the original structure of the application's code and some of its problems. Finally, it describes the way of deploying the original application and presents some concepts that modernize the process applied as a part of this work.

2.1 Requirements

The proposed extension of the application revolves around providing a unified access to graphs from collections put up on the Internet by graph experts. The unified access is realized by importing the collections into our database and allowing the users to limit themselves to work only with graphs from specific collections. The original application already provides a solid user interface which enables the users to query, view and download graphs stored in the application database. The existing UI can be used to serve the collections with only slight adjustments. Therefore no extensive work on the frontend is required, most of the changes are done on the backend.

2.1.1 Graph collections

First, let's go into more detail on what a graph collection actually is. A graph collection is a set of graphs in which all graphs share a specific distinctive feature. The collection must not contain two isomorphic graphs. Isomorphic graphs are two graphs which contain the same number of vertices connected in the same way. Formally described in Diestel's Graph Theory[1]: Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $\phi : V \to V'$ with $\{x, y\} \in E \Leftrightarrow \{\phi(x), \phi(y)\} \in E'$ for all $x, y \in V$.

The collections are typically complete, at least up to a given order. This means that there exists no graph which has the collection defining properties and which is not included by the collection. The number of collections is almost infinite, as the creator of the collection can freely combine the properties that define it. Though it may be obvious, it should also be mentioned, that a single graph can be part of many different collections.

There are a number of websites hosting various graph collections. The collection is usually a file, in which each line represents in arbitrary format vertices and edges of one graph. We typically know nothing about the graphs apart from the properties which define that collection. The extended application is able to associate the graphs with the collections they belong to. Then the users are allowed to limit themselves to certain collections and continue working only with the graphs from these collections. The original database holds a collection of all graphs up to the order of 10 vertices with some of their properties computed and stored as well. It may not look like much on the first glance, but this collection consists of 12,293,435 non-isomorphic graphs.

2.1.2 Importing collections

Now that it is more clear what collections are, it is necessary to talk about what is meant by importing a collection. As stated in the previous section, the graph collections are usually available on the web as text files containing graphs represented by various formats. When importing such collection into our application, two things can happen. First, the graphs in the collection are already present in our database with their properties computed. In this case, importing the collection means merely labeling the graphs in our system. Second, some or all of the graphs in the collection are not included in our database. In that case, they need to be added and their properties computed first.

Adding graphs to the database in the original application is a tedious process completely inaccessible to the application users. To add graphs and compute their properties, an administrator must log in to the server hosting the application. He/she must copy a file containing a list of graphs in a specific format to the machine and run an import script. Computation of some of the properties falls into the NP class of problems. Therefore, especially when dealing with larger graphs, the person doing the import needs to carefully select the properties he/she wishes to compute or he/she could get stuck on such difficult computation, as there is no timeout mechanism implemented, potentially leading to the need to stop the script and run it again without the difficult property. To address these issues, the extended application provides a proper functioning interface for adding and computing one or multiple graphs, accessible from outside of the server's filesystem.

2.1.3 Graph formats

Before actually importing any collections, exploration of the most common graph formats in the collection sources from the thesis assignment is needed. First, let's look at the graph collections available from the House of Graphs [2]. Some lists are hosted directly on the website, while others are provided as links to pages hosted elsewhere. Only the collections hosted directly on House of Graphs are considered. Each collection page gives a brief description of the collection attributes and a table with links to the individual collections

by the number of graph vertices. An example of such page can be seen in the Figure 2.1. The table also shows the total number of graphs included in the given collection. When following one of the collection links, the actual collection file contents can be seen. On each line there is a string of ASCII characters, for example:
*E~z_*
E^rG
E}vO
These strings represent a graph in the Graph6 format, invented by Brendan McKay [3]. The format is also very well described in Herbert Ullrich's bachelor thesis [4] on pages 8 and 9. The larger collections are provided as a compressed file and have a *.gz* extension.

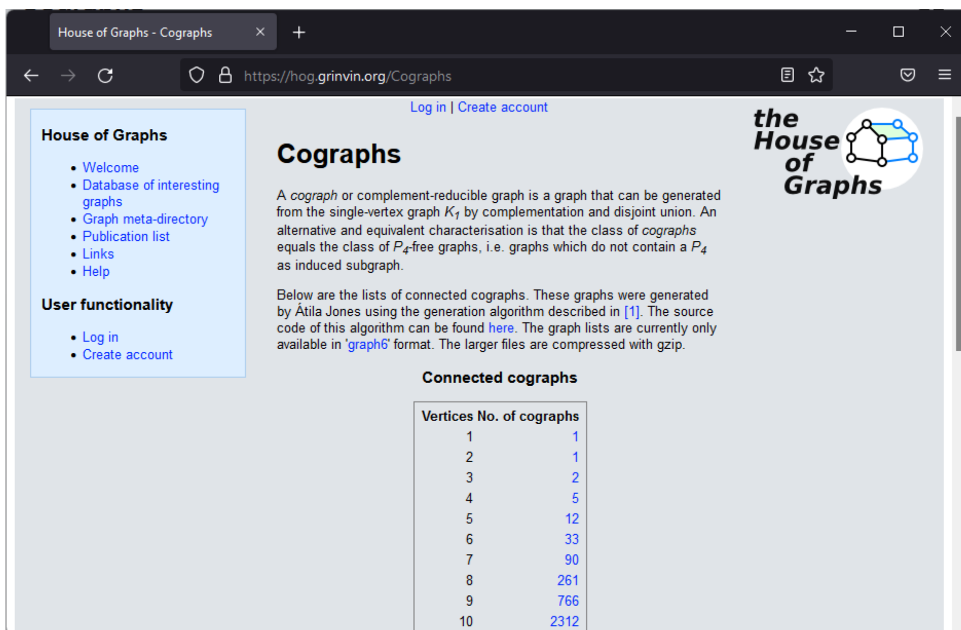


Figure 2.1: An example of collections available from the House of Graphs website

Moving on to the next collection source, the Brendan McKay's website [5]. Following the link to Brendan's data page presents us with with a simple website containing not only graph collections, but other types of combinatorial data as well. As can be seen from the Figure 2.2, the page states that unless otherwise specified, graphs are presented in either Graph6 or Sparse6 format. It also states, that larger files are compressed and have a *.gz* extension, the same way it is in the House of Graphs.

One of the other formats can be seen in the collections of trees available at `http://users.cecs.anu.edu.au/~bdm/data/trees.html`. Graphs in these collections are represented by an edge list. Each edge in the list is represented by two integers, separated by a space, denoting the vertices which the edge connects. The edges are then separated by two spaces, encoding the whole graph into one line. An example of a graph encoded by an edge list is as follows: *0 7 0 8 1 7 1 9 2 8 3 9 4 9 5 9 6 9.*

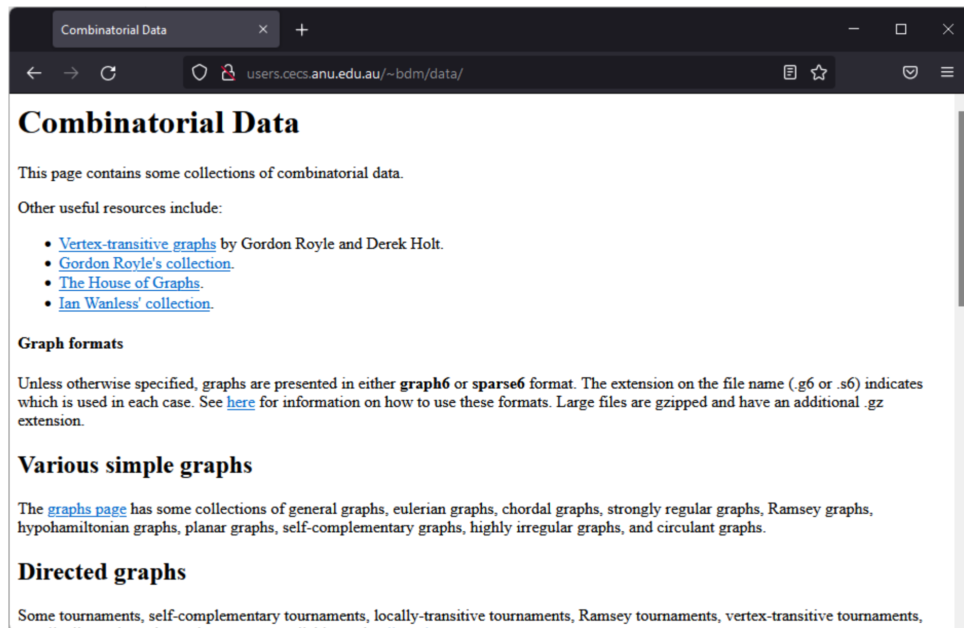


Figure 2.2: The data webpage of Brandon McKay's website

The last collection source mentioned in the thesis assignment is the Encyclopedia of Graphs [6]. The main page of the website provides a link to the list of collections hosted on the website. The collections are available for download in form of a *.zip* file containing the Sparse6 codes of the selected graphs, as can be seen in the Figure 2.3. The zip file is composed of multiple text files, one for each graph present in the collection. Each of these text files contains a string of Sparse6 representation of a graph.

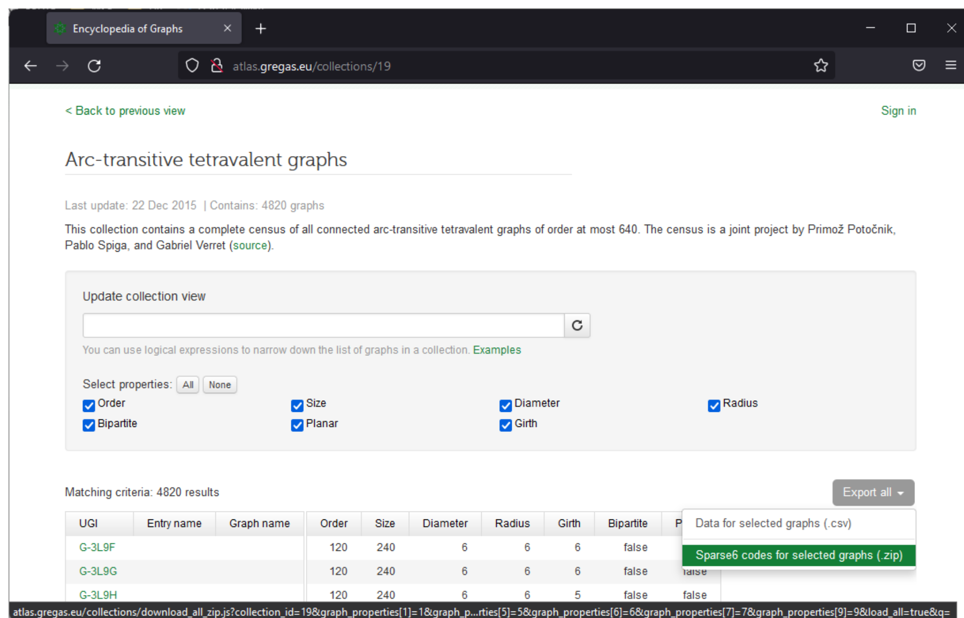


Figure 2.3: Download of a collection from Encyclopedia of Graphs

Links to the detailed descriptions of the mentioned formats, as well as other graph formats is available in the Appendix A. In conclusion, there are three possible ways of obtaining data of a collection. The extended application supports all of them:

- Downloading a text file with one graph per line
- Downloading a gzip compressed text file with one graph per line
- Downloading multiple zipped files with one graph per file

And there are three graph formats widely used accross the explored collections. The extended application can parse all of them as well:

- Graph6
- Sparse6
- List of edges

2.2 Original application structure

Next, let's explore the source code of the project. The web application is written in JavaScript. It uses the Express framework for backend and ReactJS for frontend. The database used to store the graphs and their properties is Postgres. The scripts for adding graphs into the database, mentioned in Section 2.1.2, are written in Python. There are also some other Python scripts which will be described in the upcoming sections. The source code is divided into four main parts, each residing in a specific directory.

- node-www
- node-www/compute
- db_update
- db_counter

The following sections will go into more detail on each of them.

2.2.1 node-www

This directory contains the web application's JavaScript code, both frontend and backend. The backend is based on Express. According to its documentation [7], Express is a lightweight web framework, which can be used to handle HTTP communication and thus provide an API encapsulating the communication with the graph database. The framework is also capable of serving static HTML, CSS, and JavaScript files to be interpreted by the browser. Apart from providing data from the database, the API can also serve images of the graphs, which are created using the Graphviz software. For more information about Graphviz, refer to its documentation [8]. Going into more detail on Graphviz is not relevant for this thesis, as the graph visualization functionality is left untouched.

The frontend is a Single Page Application written using the ReactJS framework. Single Page Applications typically load all the application's clientside HTML and JavaScript code when the user first visits the webpage. The JavaScript code then makes asynchronous requests to the server to fetch additional data. Based on the received data it dynamically updates the UI presented to the user. This allows for a very fluent user interface because the browser does not need to make new requests every time the user navigates the screens inside the application. However, the downside of this approach may be a longer initial loading time of the website. The response to the first request tends to be quite large, as it usually contains the whole application.

According to the documentation [9], React allows to create reusable components. Each component can have its own state which might modify its appereance. The components can be then combined to create the UI as a whole. The framework takes care of which components to render based on the user's webpage navigation and inputs. The components can be written either as classes or as functions. The return value of the functional component or the render method of the class component describes what the UI should look like. This is usually done by JSX, a syntax extension to JavaScript. JSX uses HTML tags with embedded JavaScript code.

Before being served to the clients, the frontend code is bundled using Webpack. This allows the React components to be "compiled" into vanilla JavaScript, easily interpretable by most modern browsers. The result of this is a single JavaScript file containing all the clientside code. The bundle does not contain human readable code, as it is usually condensed into a single line to save as much space as possible, resulting also in time being saved when the file is transported over network. The version of Webpack has been upgraded in the extended application to support newer versions of JavaScript server-side runtime. More information about Webpack can be found in its documentation [10].

The application UI contains a page showing the user an option to insert a graph and compute its properties. However, due to the implementation of adding graphs in the original application, discussed in Section 2.1.2, the page does not work and when the user tries to insert a graph, a loading spinner appears and never finishes. This is captured in the Figure 2.4 and it leads to a very poor user experience. The Section 3.7 describes how is this solved in the extended application using a proper functioning API.

2.2.2 node-www/compute

The subdirectory of the web application contains the logic for computing graph properties using different mathematical libraries, either Sage [11], NetworkX [12] or Wolfram [13]. One of the original backend API endpoints (*/api/graph*) was to allow you to choose which properties and library to use to calculate a given graph. However, the endpoint does not work, causing the infinite spinner mentioned in the Section 2.2.1. This functionality was probably superseded by the manual graph adding described in the Section 2.1.2, which partly reuses the code for Sage calculations.

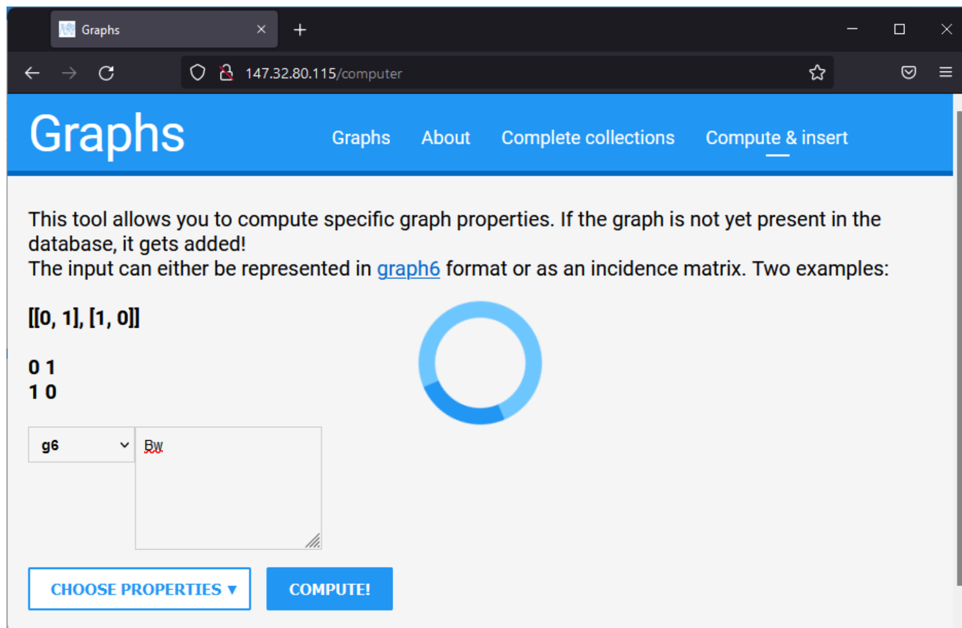


Figure 2.4: A non-functioning page for inserting new graphs into the database

2.2.3 db_update

This Python module encapsulates some of the database management, already touched upon in the Section 2.1.2. Mainly adding new graphs and computing their properties. The scripts are supposed to be executed via a bash script, which is used to parse and validate commandline arguments. To compute the properties, they import some of the code from the Sage computation engine residing in the *node-www/compute* directory.

The way the scripts communicate with the database is very questionable to say the least. The Python code does not connect to the database directly using a Postgres database adapter. Instead, for each database query, it spawns a subprocess of *psql*. Cited from the documentation [14]: "psql is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results". After spawning the subprocess, the scripts then send the SQL query as a string to the standard input of the subprocess and parses the results from its standard output. This approach is flawed on multiple levels. Even though SQL injection does not concern us here because the queries are being run by the administrator, the usage of raw strings for representing database queries is still considered bad practice. Writing queries as strings is very prone to error and it makes the code less readable and maintainable. Another issue with this approach is the performance impact. Creating a new process as well as establishing a new connection with the database for each query is very costly. This is in contrast with the over-optimized queries the original author has created, including dropping the index of the updated columns or creating temporary tables for faster insertions, which further decrease the code readability. This is fixed in

the extended application by using the SQLAlchemy library, which is described in the Section 3.4.

One of the requirements in the original project specification was for the database to be extensible for the calculation of new graph properties. It can be extended, but again, it is quite a lengthy process. To extend the database and compute new properties, the database schema must be manually edited by connecting to the database and running an *ALTER TABLE* SQL command. Then it is needed to add the property to the Sage calculation definitions in *node-www/compute*. And finally a JSON file containing all the column definitions must be edited in the *db_update* directory, before running the update script to compute the new property. Although it works, there is still room for improvement. The extended application improves this by incorporating database migrations, described in the Section 3.5, and the implementation of the computation engine, described in the Section 4.1.4 and the Section 4.1.6.

2.2.4 db_counter

This is a set of Python scripts that also attempt to enter some kind of information about graph collections into the database. The scripts run a set of queries that check the number of selected graphs present in the database against a verified source providing the number of graphs belonging to the collection. Based on this it claims that the whole given collection is included in the database and stores the information in a separate database table. However, it provides no further interaction with the graphs or the collections, it simply states that it is there. These scripts also need to be run manually on the server.

The original UI contains a page showing the data created by these scripts. The page can be seen in Figure 2.5. This feature provides users with only limited value in the form of static information about the underlying graph database. The fact that the description visible in the user interface ends in the middle of a sentence also suggests that it may be an unfinished concept. This problem is solved by the removal of this whole feature from the extended application, described in the Section 3.2.

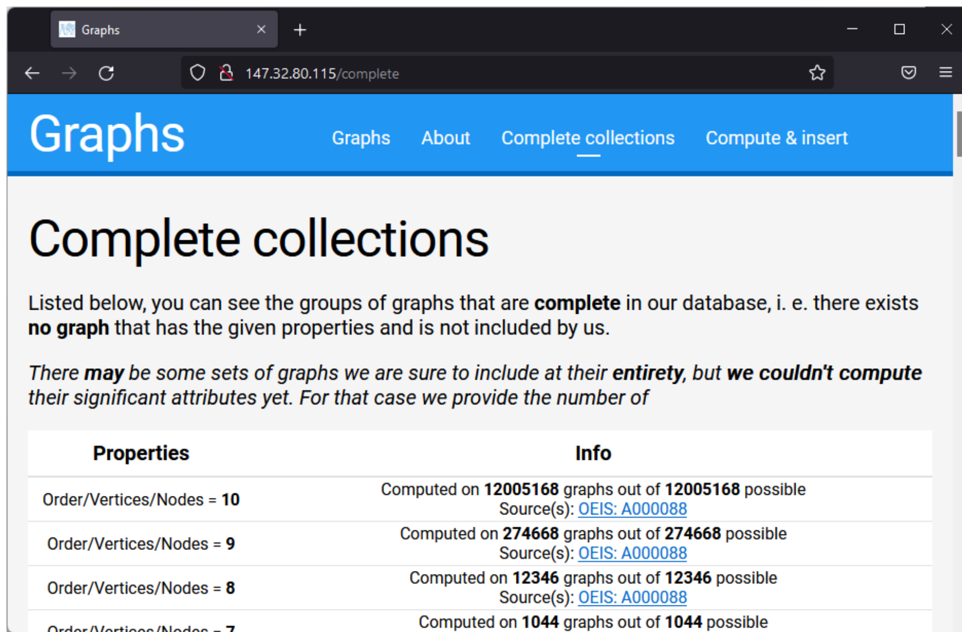


Figure 2.5: Complete collections page in the original application

2.3 Deployment

After examining the source code, we can take a look at the deployment process of the original application. The application as well as the database is deployed on a virtual machine provided by the CTU IT Centre [15]. The machine contains a cloned copy of the project's git repository. The deployment is done by manually logging in to the server, pulling the latest changes from git, repacking the frontend code and finally restarting the whole JavaScript application. The host system environment has been set up manually by the authors of the original application during the development.

The environment needed to run the application is quite hard to recreate, as it requires other external software installed on the system. However, there is no single source specifying the needed software and its configuration. One must go through the multiple *README.md* files present in the repository, to find some of the information. Reading the documentation provided in these markdown files is not enough, because some of them are extremely outdated and even the ones up to date do not convey all of the needed information. Some of it can only be found by exploring the application's configuration files, the source code and by running the application until it fails with an error. The complete list of external dependencies is following:

- nauty
- Graphviz
- Sage
- g++

- make
- psql

The exact database schema is also nowhere to be found in the project's source code or configuration. It has been created during the development and set up directly in the production database on the server. The most efficient way to recreate the database locally for development purposes that I found was saving the database schema using the *pg_dump* [16] utility and then executing the generated SQL script against the local development database.

The untracked database schema together with the complex list of dependencies makes it difficult to deploy the application in a new location, or to set up a working development environment. It requires carrying out many undocumented manual tasks which, without prior thorough knowledge of the whole system, may lead to many "trial and error" situations.

2.3.1 Dockerized applications

The problems stated in the previous Section can be solved by the concept of dockerized application. Dockerized application is packaged into a standardized unit called a container. Containers contain everything the application needs in order to run properly - libraries, software, system tools, the application's code, and runtime. Docker [17] is then able to build and run these containers. Thanks to this, none of the application dependencies need to be installed directly on the machine on which is the application supposed to run, be it a production server or a developer's workstation. All that is needed is Docker.

The main building block of dockerized applications is the so-called Dockerfile. Dockerfile serves as a "recipe" for getting the dependencies, building and finally running the application. The recipe is not only useful for the Docker engine, but for the developers as well. If it is needed to actually run the application directly on the developer's system, for example for debugging, the information about the dependencies is easily obtainable from the Dockerfile, as it is composed of simple commands needed to set up the container. The Dockerfile is versioned in the git repository together with the code, making it easily accessible and also tied to the version of the application, should the dependencies change over time.

Docker OS limitations

The containers can be based on various Linux distributions, not limited to the one running on the machine. Docker then uses the Linux kernel of the host OS to simulate other Linux distributions inside the containers. Each container is run as a separate process on the host OS, while Docker uses features of the Linux kernel (namely namespaces and control groups) to isolate these containers from one and other for added security.

The fact that Docker needs Linux kernel to run does not mean that it cannot be used on Windows or Mac. If virtualization is enabled on these systems, the corresponding versions of Docker run in lightweight virtual

machine. On Windows 10, version 1903 or later, Docker can even use the virtualized Linux kernel provided by Windows Subsystem for Linux (WSL). All of this is completely transparent to the user. Moreover, the Docker for Windows can also run containers based on Windows images, which is ironically not possible on a machine running Linux.

2.3.2 CI/CD

Dockerizing the application helps us to firmly define the application's dependencies and make it easily run anywhere. However, it does not change the fact that in order to deploy or update the application, the administrator still needs to connect to the server, pull the latest changes and rebuild the container. That's where CI/CD comes into play. CI/CD stands for continuous integration and continuous delivery. It is typically realized by automated pipelines, which execute predefined steps each time there's a new revision of code in git or before and after a branch is merged.

Continuous integration automates the process of building, packaging, and testing the application. The project contains a good amount of automated tests written by Tomáš Roun as a part of his Bachelor thesis [18]. I think it's safe to say that there is no one who never forgets to run tests before commiting a change or merging a branch, and so the execution of the tests also needs to be automated.

Continuous delivery starts where continuous integration ends and it is the key to solving the remaining problem. As described earlier, dockerized applications are extremely easy to get running anywhere. This allows for trouble-free automation of the deployment process. This automated deployment can become a part of the CI/CD pipeline provided the tests are executed successfully. Dockerization and CI/CD go hand in hand and help to make the developers' lives easier by automating the manual work associated with developping applications and allowing them to focus on writing code.

Chapter 3
Design

The previous chapter explored the requirements of the extension and the structure of the original application. Based on the performed analysis, it is necessary to design the application extension and choose the technologies that are used for the implementation. This chapter describes the flow of importing collections, the structural changes of the project, the changes regarding the database and the communication with it, the selection of an appropriate web framework, the API design and finally the automated deployment.

3.1 Importing collections

As mentioned at the end of the Section 2.1.2, the extended application provides an API for importing collections and their graphs into our database. However, we do not want to permit anyone adding any collections. There still needs to be some kind of supervision over what should and what should not be imported. Because of that, the extended application now has two roles - users and administrators. The intended functionality of the original application is available to the basic users. In addition to that, the users can suggest a collection to be imported by submitting a name of the collection and a link to the collection file. The administrator then must resolve these suggested collections, either by approving or rejecting them.

The lifecycle of the collection is best described by the swimlane diagram in the Figure 3.1. The collection can be in one of these five states:

- SUGGESTED
- REJECTED
- APPROVED
- INVALID
- IMPORTED

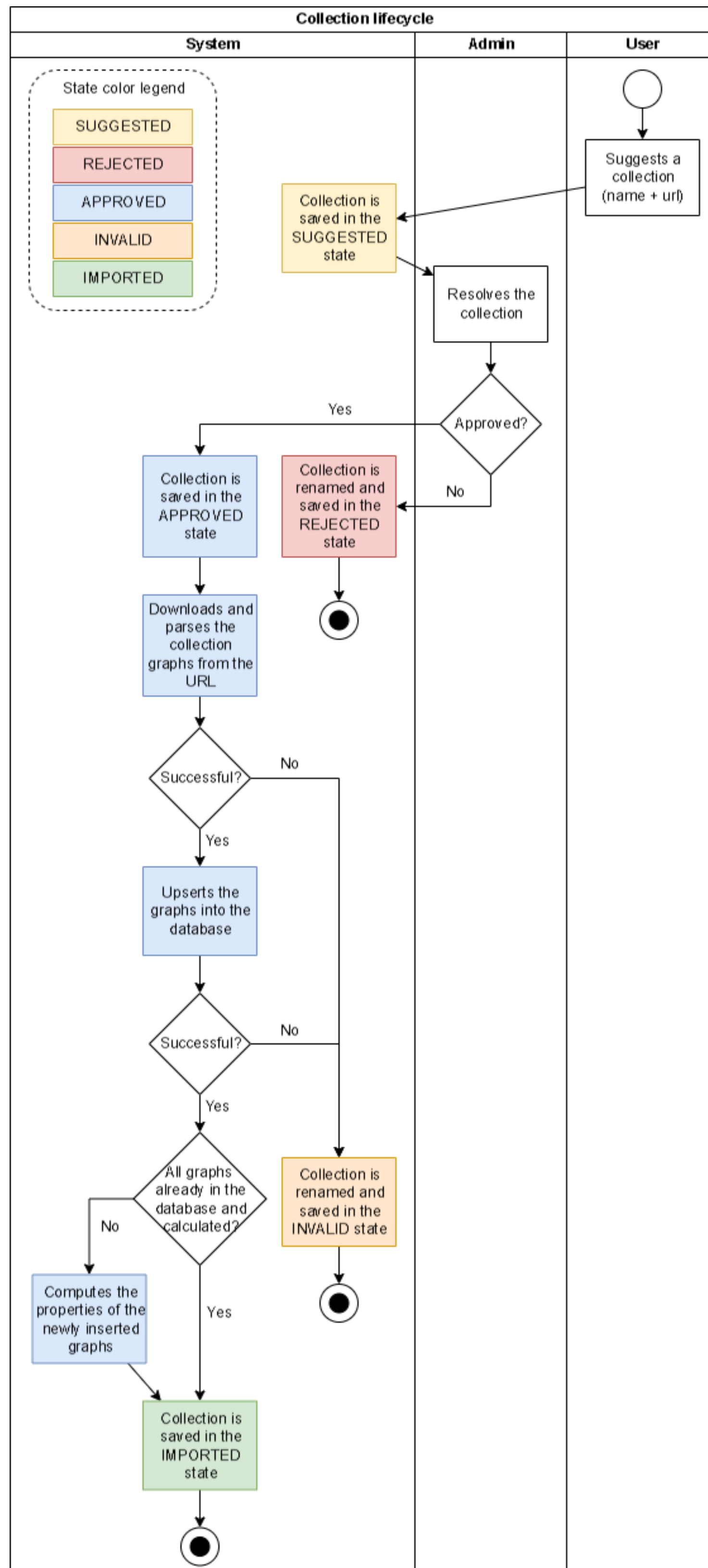


Figure 3.1: Collection lifecycle swimlane diagram

For better clarity, the colors of the nodes belonging to the system swimlane represent the collection state during that step by their color. The legend of the colors can be found in the top-left corner of the diagram. One of the nodes uses the word *upsert*. This basically means *update or insert*. In the Postgres database, this is achieved by the *ON CONFLICT DO UPDATE* clause of the *INSERT* statement. Lastly, the diagram shows, that the collection can get into the invalid state either by the failure of downloading and parsing the collection or by the failure of upserting its graphs into the database. The former is the case if the collection cannot be downloaded from the provided link or when the collection contains graphs in a format unknown to our application (the supported downloadable file and graph formats were presented at the end of the Section 2.1.3). The latter reason to invalidate the collection is tied to the limits of the database. The insertion can fail for graphs whose Graph6 representation does not fit to the column protected by a unique constraint in our database. In Postgres, a unique constraint is implemented with a unique B-tree index. Every index is stored as an array of data pages of a fixed size of 8kB. The maximum size of an index entry is a third of a data page, which is approximately 2730 bytes.

3.1.1 Administrator access

When implementing the administrator access, the goal was to make it as simple as possible for the administrators as well as for the system itself. That is why it was decided to integrate with the CTU single sign-on (SSO), which is used for example to log in to KOS. This way our application does not need to manage a user database and the administrators do not need to remember special credentials for our application.

After consultation with Ing. Votava from the CTU IT Centre, who is responsible for the SSO authentication, it was decided to integrate our application via the OpenID Connect protocol. According to the documentation [19], the protocol works with three actors:

- Client - in our case the Graphs application itself
- Resource Owner - in our case anyone visiting the Client, reffered to as "user" in the following paragraphs
- Authorization Server - in our case the CTU SSO service

First, the Client was registered in the Authorization Server and specific people from the CTU domain were granted the admin role within the context of the Client. This provides us with a *client_id* of our application. Now, whenever a user (Resource Owner) wishes to log in, he/she is redirected to the */auth* endpoint of the Authorization Server. The previously mentioned *client_id* is a part of this request. The request also contains a *redirect_uri*, which points back to the Client. After the user successfully logs in, he/she is redirected by the Authorization Server back to the *redirect_uri* with an authorization code. The Client then calls the */token* endpoint of the Authorization Server to exchange the authorization code for an authorization token. This is a JSON

Web Token (JWT), which contains some of the basic information about the user for whom it was issued by the Authorization Server.

This token is sent with the requests to our API endpoints which require administrator access. The token contains information about the user's roles within the Client application's context. Based on this, the backend validates, whether the user can access the endpoint or not. To ensure the users cannot modify or forge the tokens, they are cryptographically signed by the Authorization Server. The Client backend validates the signature using the public key of the Authorization Server, which is freely available.

3.2 Extended application structure

After dissecting the original source code structure in the previous chapter, a new, simplified structure is applied to the project. The proposed extension adds the ability to work with graph collections in a much more dynamic way, than currently provided by the *db_counter* module, described in the Section 2.2.4. Therefore, after a consultation with the supervisor, it was decided to remove the whole concept from the application. This includes removing the *db_counter* scripts, the corresponding database tables and the unfinished UI page.

The *db_update* module is refactored into a full-fledged service overseeing the graph additions. The code for Sage calculations from *node-www/compute* becomes a part of the new service. The non-functioning API endpoint as well as the other computational engines described in 2.2.2 are discarded, because they were not in use. The new service serves its own API providing all the functionality needed for managing the graph database, further described in the Section 3.7. To keep things simple on the frontend, the client application still communicates only with the Express backend that is serving it. The backend acts as a gateway, forwarding some of the requests to the new service API.

The new service is written in the Python programming language. Python is an interpreted scripting programming language. It is a hybrid language, which means that the program does not have to be all object-oriented, but parts of the program can be procedural. This contributes to better code readability and overall simplification. Python is the language of choice for its easy integration with Sage, the option to reuse some of the code from the manual graph insertion scripts and lastly for the personal preferences of the author.

3.3 Database schema

To implement the flow described in the Section 3.1 a new table is added to the graph database. The table holds the collections and their state. The table is quite simple, as can be seen in the diagram in the Figure 3.2, it has only three columns - name, url and state.

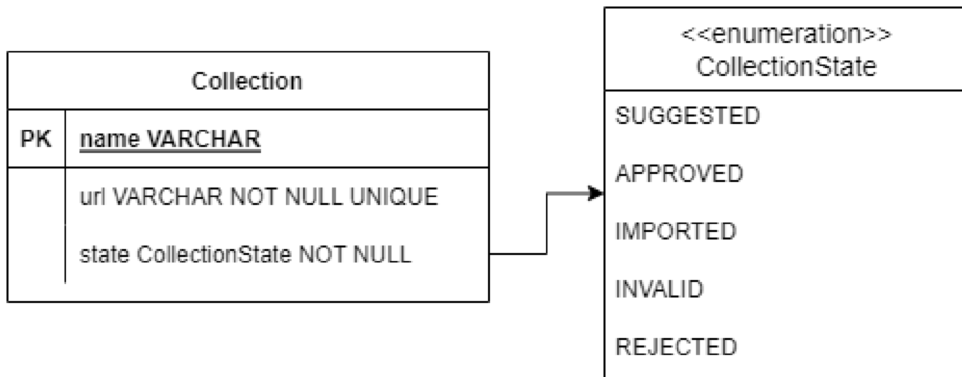


Figure 3.2: Collection table database schema diagram

Some changes are done to the graph table as well to support the extension. As mentioned in the previous chapter in Section 2.1.2, the computation of the properties in the original application implements no timeout mechanism. This was not a real issue, as the computation was being triggered manually and watched over by an administrator. However, because the extended application now supports adding graphs from collections automatically, such timeout mechanism is implemented. To distinguish the graphs with properties, whose computation timed out, from the graphs, that were added into the database, but their computation has not even started yet, a boolean column *computed* has been added to the table.

To associate the graphs with the collections they belong in, a *collections* column has been added to the table as well. The column is of type *ARRAY* and contains strings of the names of collections. This approach has been chosen for its simplicity, as the array column supports the *overlap* (&&) operator, which is all that is needed to filter graphs from a specific combination of collections.

3.4 Communication with the database

As described in the previous chapter in the Section 2.2.3, the original application uses a very questionable way to communicate with the database. Another problem that section describes is the absence of any database schema tracking, possibly versioned together with the source code. The extended application kills two birds with one stone by incorporating an ORM layer in between the database and the application logic.

ORM stands for Object-Relational Mapping. Essentially this means mapping the relational schema of a database into classes of a given programming language. A table in the database then corresponds to a class definition, columns in the table correspond to the attributes of the class and finally the actual rows can be represented by the class instances. As the definition of these classes becomes a part of the application, the schema is now versioned in git accordingly.

The ORM classes are not only used to define the database schema. The libraries implementing the ORM layer usually provide an abstraction of some basic SQL queries. This abstraction may include methods on the ORM class instances like *.save()* or *.delete()*, which internally invoke an *INSERT* or *DELETE* SQL statements, or static methods like *.getById()*, which runs a *SELECT* statement and returns the row as an instance of the class. This abstraction is really easy to work with. However, its uses are limited, as some of the more complex queries involving table joins, advanced filtering or operations on a large number of rows cannot be accomplished with ORM in a well performing manner, or in some cases at all.

3.4.1 SQLAlchemy

As stated in the previous section, querying database using ORM is very limited. ORM really shines when working with individual table rows, like fetching an entry as a class instance, modifying its attributes and saving it back to the database. However, this is not what is usually done in our application. Typically many graphs are being inserted or updated at once. That is why SQLAlchemy is the library of choice for the new service of the extended application. Cited from the documentation [20]: "SQLAlchemy consists of two distinct components, known as the Core and the ORM. The Core is itself a fully featured SQL abstraction toolkit, providing a smooth layer of abstraction over a wide variety of DBAPI implementations and behaviors, as well as a SQL Expression Language which allows expression of the SQL language via generative Python expressions". This means that the ORM classes can be used to track the database schema, while the full feature set of pure SQL is still accessible, but in much safer and more maintainable way than hardcoding SQL strings, as it was in the original application.

Another great feature provided by SQLAlchemy is its event system. It allows for Python functions to be registered as callbacks for specific database events, like inserts, updates, session commits, etc. This is used for communication between parts of the new service. There are two use cases for this in the extended application. First, when new graphs are inserted into the database by the insert module, the event hook of the update module picks up on that and schedules a computation of the properties of the new graphs. The other is when the update module finishes the computation of graphs properties and updates them in the database, the collection module picks up on the event and checks, whether it can move any approved collection into the imported state, as described in the Section 3.1.

3.5 Database migration

Even though the database schema is now versioned together with the code via SQLAlchemy's ORM layer, it only allows to set up the schema in a clean database. But when the ORM specification changes, it does not affect an existing database schema. To change an existing database schema, a database

migration must be carried out. Database migration is typically defined as a set of DDL commands (Data Definition Language - a category of SQL queries) like *CREATE TABLE*, *ALTER TABLE* or *DROP TABLE*. SQLAlchemy recommends using Alembic for this purpose.

Alembic is a tool that can partially automate the database migration process. By comparing the current schema of the database and the ORM definition of SQLAlchemy, the tool is able to generate migration scripts, which contain the DDL commands needed to transfer between the two states. Alembic also keeps the individual schema states as revisions, which in principal work similarly to the git revision system. As stated in the Alembic documentation [21], the migration script generating is not intended to be perfect. It is always necessary to manually review and correct the scripts that autogenerate produces.

3.6 Web framework choice

Because the new service provides a web API, it is important to choose a suitable web framework. Apart from providing the API, the service acts as a computation engine for newly added graphs. Therefore, the chosen framework must perform well to save resources for the computation engine. There exist a number of web frameworks for Python. We consider only some of the well-known ones. The popular frameworks usually have a large community behind them, which makes it easier to work with them. If we run into a problem, there is a high chance that someone else has already encountered and resolved a similar issue before. We can then draw from their resolution to help ourselves.

This section now looks in more detail on some of the selected frameworks and clarifies the choice for this project.

3.6.1 Django

Django [22] is a fairly complex framework providing a lot of features out of the box, like application administration or database management. It is common for Django project to also create and serve the UI. Django follows the Model-Template-View (MTV) pattern, which is a slight modification of the Model-View-Controller (MVC) pattern.

MVC is a software architecture pattern which separates data storage and access (Model), the presentation of the data and receiving the user input (View), and the logic of handling the user interactions (Controller). The flow is usually following:

1. The user sees the View and interacts with it
2. The View sends the input to the Controller
3. The Controller performs the application's business logic
4. The Controller uses the Model to modify the data
5. The View gets updated based on the new data

MTV is a little different. Model in the MTV pattern is the same as in MVC, as it encapsulates the access to the database. Template is closest to the View in the MVC pattern. It is the presentation layer that controls what should be displayed and how it should be displayed to the user. View in the MTV pattern relates to the Controller and partly to the View from MVC. It contains the business logic of manipulating the data and handles the user interaction by receiving the input and responding with the templates filled with the appropriate data. This can be seen in the diagram in the Figure 3.3. Django typically groups the code that handles each of these steps into separate files.

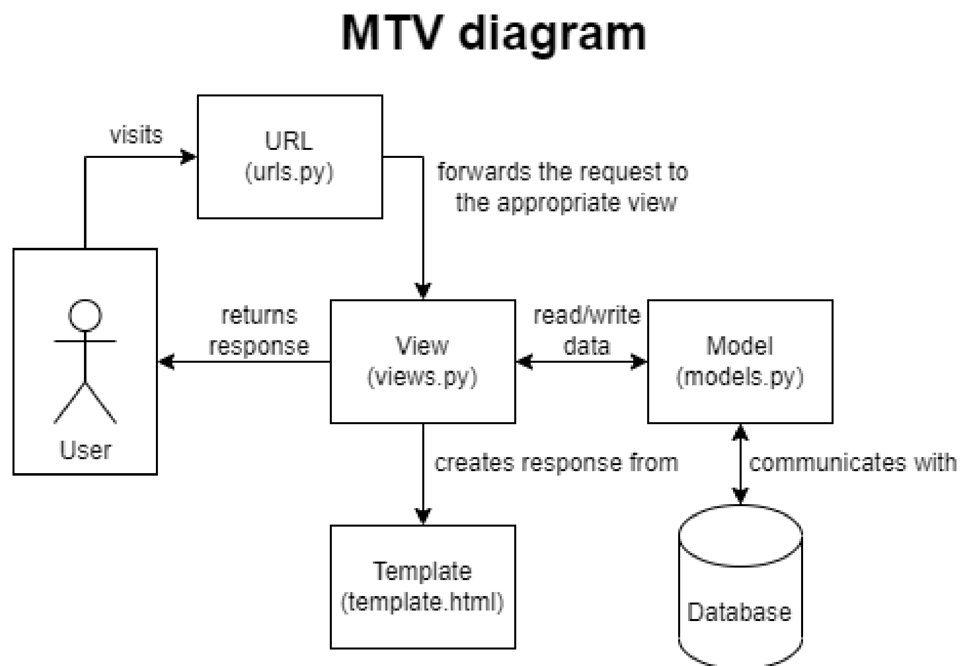


Figure 3.3: Model-Template-View diagram

Django comes with an administration commandline tool *django-admin*. The framework enforces quite strict project structure, which is automatically generated by the administration tool when creating a new project. Django is great for making systems revolving around CRUD operations (Create, Read, Update, Delete) of several resources provided to the users, as most of the functionality comes out of the box with just setting up the project.

3.6.2 Flask

Flask [23] is a minimal web framework, in a sense similar to JavaScript's Express. Minimal meaning that the core of the framework is simple, but extensible. By default, it does not provide any additional features out of the box like Django, but they need to be supplied by other libraries or written from scratch. It also does not force a specific project structure. In an extreme

case, an application can fit into a single Python file. This of course is not the recommended approach.

The Figure 3.4 shows an example of an API endpoint created with Flask. The endpoint is created by annotating a function. The annotation specifies the URL and the HTTP method. Flask is implicitly synchronous, meaning it does not allow requests to be handled concurrently. Concurrency is achieved by using Flask with a WSGI [24] server, which handles the requests in threads. To access the data of the request, like the headers or the body, a *request* object must be imported from *flask*. This is internally implemented as a thread-local object, exclusive for each request that is being handled. This object does not need to be passed between functions and can be imported anywhere within the request.

```
from flask import abort, jsonify, request

# app is an instance of Flask
from my_project.some_path import app
# payload validation logic
from my_project.other_path import is_valid

@app.route("/compute", methods=["POST"])
def compute():
    payload = request.json
    if not is_valid(payload):
        abort(400)
    # handle the request and create a response
    return jsonify(response)
```

Figure 3.4: Example of an API route definition using Flask

Flask is not meant to be used as a black-box framework. For larger projects it is encouraged to read and understand the implementation of the framework, as it offers a large number of available overrides, hook points or signals. Custom classes can be provided for various things, for example like the request and response objects.

All in all, Flask gives much more freedom to the developer than Django, at the cost of the out of box features. The loss of the features is not inherently a bad thing, as many of them may not be of use for the particular project anyways. Setting up API endpoints can be accomplished in a matter of minutes. This makes Flask great for small applications. Building large applications with Flask requires a certain level of expertise, as the unrestrained project structure can more easily result in a messy source code.

3.6.3 Connexion

Connexion [25] is a framework built on top of Flask. However, it removes the need to write any code associated with defining the URL routes, the HTTP methods of the endpoints, etc. This is achieved by the framework by loading an OpenAPI specification [26] and automatically mapping the endpoints declared in it to Python functions.

OpenAPI is a documentation format used for defining API endpoints, their parameters, different response types and payloads. The specification can be written either in YAML or JSON format. An example of such endpoint definition in YAML can be seen in the Figure 3.5. The specification can be divided into multiple files which reference one and other using the *$ref* keyword. Thanks to this, even very large API specifications can be managed comfortably.

```yaml
post:
  tags:
    - collection
  summary: submit a new collection for approval
  operationId: submit
  requestBody:
    required: true
    content:
      application/json:
        schema:
          $ref: "../schemas/collection.yml#/Collection"
  responses:
    "204":
      description: success, no content
    "400":
      description: invalid collection entity
      content:
        application/json:
          schema:
            $ref: "../schemas/message.yml#/Message"
    "409":
      description: collection name or url already exists
      content:
        application/json:
          schema:
            $ref: "../schemas/message.yml#/Message"
```

Figure 3.5: Example of an API endpoint defined in OpenAPI

Apart from mapping the endpoints to Python functions, the framework also validates the data sent by users and automatically responds with an

appropriate HTTP code and message when something is not correct. OpenAPI also allows to generate Python model classes for the payload types present in the specification. This makes for more convenient data manipulation inside the application. Connexion also automatically serializes instances of these classes when they're returned by the endpoint functions.

Although the YAML or JSON specifications are very human readable, OpenAPI can even be used to generate a web UI displaying the endpoints in an extremely user friendly way. This UI even allows to call the endpoints with arbitrary parameters and see the responses. Connexion can deploy such UI automatically, if configured so.

In summary, Connexion forces the developer to write the specification first, which ensures that the API documentation is always up-to-date. It handles route creation, data validation and response serialization. It can provide an interactive web UI to view and call the endpoints. All that while keeping the freedom of the rest of the project structure like Flask does.

3.6.4 FastAPI

Working with FastAPI [27] is in a way similar to Flask, by using function annotations to define API routes. However, it also provides automatic data validation and responses just like Connexion, without the need to first write the specification. This is done by the framework by parsing Python type hints.

Python is a dynamically typed language. This means that the type of objects stored in variables is checked during runtime. While this allows for more flexibility, it is more error prone. The developers need to make sure they are using the correct types when passing arguments to functions, performing arithmetics etc. Type hinting helps with that by annotating variables, arguments and return values with their expected type. An example of the usage of type hints can be seen in the Figure 3.6. This function is supposed to take in a string argument, an integer argument and return a boolean. However, type hinting in no way changes the execution of the code. The function could still be called with different types and throw an exception. But thanks to the type hints, this can be more easily caught by static code analysers.

```
def order_beer(name: str, age: int) -> bool:
    if age < 18:
        print("I'm sorry, but you can't order a beer")
        return False
    print("Here's a beer for " + name + "!")
    return True
```

Figure 3.6: Example of a Python function using type hints

FastAPI also uses the type hints to generate an OpenAPI documentation

of the endpoints and can deploy the interactive web UI same as Connexion. The result is very similar, but instead of writing specification and generating code from it, this works the other way around. The code is written first and the specification is generated from it.

Concurrency in FastAPI is not achieved by threads, like in the previous frameworks. Instead, the framework is based on the *asyncio* Python library. The core of each *asyncio* application is the event loop. Event loops schedule and run so-called coroutines. A coroutine is basically a function, that can *await* various calls, mostly associated with I/O. This includes filesystem or network I/O operations, database queries or waiting for a subprocess to finish. When a coroutine *awaits*, the control of the code execution is given back to the event loop and it can run other scheduled coroutines until they *await*, and so on. This sounds a lot less performant than using threads, but it is not, because threads in Python do not actually run in parallel. Python threads are limited by something called the Global Interpreter Lock (GIL). Any thread that wants to run needs to first acquire the GIL in order to execute code using the Python interpreter. Because of this, the interpreter is just "passed around" the threads in between the execution of single instructions, resulting in concurrency, but not true parallelism.

To summarize it, the framework offers features regarding documentation and automation of tasks similar to the Connexion framework. The big difference between them is that FastAPI is based on *asyncio* coroutines instead of threading. Asynchronous coroutine programming sometimes requires a different approach than standard synchronous programming which uses threads and is sometimes easier to do incorrectly, as blocking calls inside a coroutine may result in the whole program being blocked.

3.6.5 Result

The previous subsections introduced the web frameworks considered for building the new service of the extended application. As stated in the first paragraph of this section, performance of the framework plays an important factor. To compare the performance of the frameworks we can use a benchmark performed by TechEmpower [28], which includes FastAPI, Flask and Django. Connexion is not included, but since it is built on top of Flask, we can assume that their performances are very similar. According to the benchmark, FastAPI outperforms Django and Flask in all tested categories, and is the clear winner.

To better visualize the choice of the framework, the Table 3.1 summarizes the features, discussed in the prevous subsections, sorted by importance - more important on the left (importance subjective to the author). Based on this overview, Connexion and FastAPI provide the most features. Therefore, taking into account the performance benchmark, FastAPI has been chosen for the implementation of the new service.

Web framework features				
	Automatic request validation	Automatic API documentation	OOB database management and admin access	Free project structure
Django	NO	NO	YES	NO
Flask	NO	NO	NO	YES
Connexion	YES	YES	NO	YES
FastAPI	YES	YES	NO	YES

Table 3.1: Web framework features sorted by importance (more important on the left)

3.7 API design

When an appropriate framework has been selected, the API of the new service can be designed. The API provides access to inserting a single graph to enable a fix of the non-functioning page described at the end of Section 2.2.1. In addition to that, it provides two collection endpoints available to unauthorized users and two endpoints for managing collections only available to administrators.

3.7.1 POST /compute

This is a working implementation of the */api/graphs* endpoint of the original application. It is used when a user attempts to add a graph into our database via the Compute & insert page in the UI. The endpoint requires the request body to be a JSON object containing two attributes:

- g6 - Graph6 representation of the graph to compute
- properties - List of names of the properties to return back

The response is one of the following:

- 200 OK - Graph was already present in the database or was added and computed in time
- 202 Accepted - Graph was added into the database, but has not been computed within a time limit
- 422 Unprocessable Entity - The requested Graph6 is invalid

3.7.2 GET /collection

This is the first collection endpoint accessible to unauthorized users. It returns the collections as specified in the Section 3.3. It is used in various places by the frontend, for example when fetching the collection options for filtering

graphs, in the About page to display the imported and approved collections, and in the collection management view. The endpoint has one optional query parameter:

- state - a list of collection states to filter for, available values:
  - suggested
  - approved
  - imported
  - invalid
  - rejected

The response is one of the following:

- 200 OK - The collections are returned
- 422 Unprocessable Entity - Incorrect state value

3.7.3 POST /collection

The other collection endpoint accessible to unauthorized users. This is used to suggest collections for import. The endpoint requires the request body to be a JSON object containing two attributes:

- name - Name of the collection, must be between 3 and 100 characters long
- url - URL leading to the collection file

The response is one of the following:

- 204 No Content - The collection has been successfully suggested
- 409 Conflict - The collection with such name or URL already exists
- 422 Unprocessable Entity - The name is not between 3 and 100 characters long or the URL is not in a valid URL format

3.7.4 GET /collection/{name}/{resolution}

This is the first endpoint accessible only by administrators. The variables are provided in the query path.

- name - Name of the collection
- resolution - Available values:
  - approve
  - reject

The response is one of the following:

- 202 Accepted - The resolution has been acknowledged and collection moved to the approved state, the system will determine, whether the collection gets into the invalid or imported state later
- 401 Unauthorized - The authorization via JWT failed
- 404 Not Found - The collection with such name does not exist
- 422 Unprocessable Entity - Invalid resolution type

3.7.5 DELETE /collection/{name}

The other collection endpoint accessible only by administrators, used to delete a collection. It also deletes graphs from the database, which are not yet computed and belong only to the specified collection. The other graphs from the collection are kept in the database, but the collection tag is removed from them. The variables are provided in the query path.

- name - Name of the collection

The response is one of the following:

- 204 No Content - The collection has been deleted
- 401 Unauthorized - The authorization via JWT failed
- 412 Precondition Failed - The collection has been approved, but the system has not finished downloading, parsing and adding its graphs, therefore the delete is not possible yet

3.8 Manage collections UI

To provide the management functionality via the web UI, a simple page is added to the frontend application. The page is accessible only to logged in administrators and provides the necessary controls which call the management API endpoints described in the Section 3.7. A wireframe of the page can be seen in the Figure 3.7. The UI is very straightforward, providing a list of collections of each collection state with the control buttons available to it.

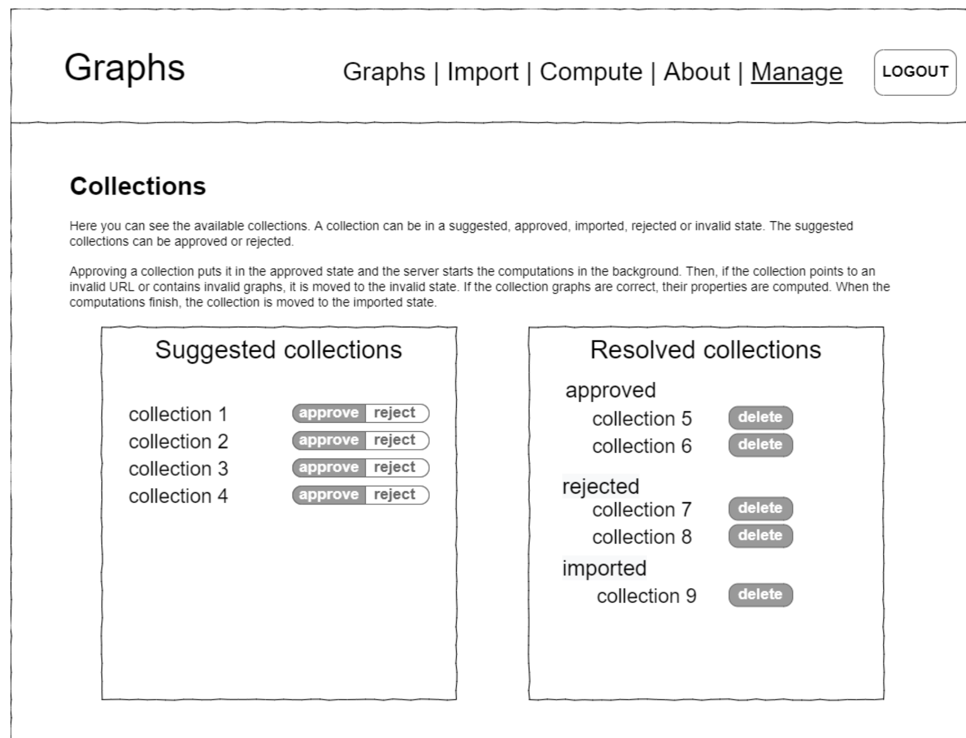


Figure 3.7: Wireframe of the Manage page in the frontend application

3.9 Deployment

GitLab Pipelines are used to automate the deployment process as described in the previous chapter in the Section 2.3.2. Each pipeline is comprised of jobs, which describe what to do - for example to build or to deploy the application. The jobs are divided into stages, which define when the jobs are run - for example jobs from the deploy stage are executed only after the build stage is finished successfully. The jobs are executed by runners, which pick up the jobs from scheduled pipelines automatically. Typically, the jobs are executed inside a Docker container, not directly on the runner. This allows execution of multiple jobs on one runner without them affecting one another. The CTU instance of GitLab provides four runners, which can be shared across projects. These are used by this project as well. The jobs inside one stage can run in parallel if there are enough free runners to execute them concurrently. The implementation of the pipeline is discussed in the Section 4.4.

GitLab is also able to securely store secrets and other variables of each project and inject them to the job's environment before its execution starts. This increases the application's security, as the production secrets, like for example database usernames and passwords, do not need to be managed manually or worse, stored in the repository together with the code.

From the point of view of deployment, the extended application is divided into three parts that are built and deployed separately - the database, the JavaScript backend that also serves the UI (referred to as *web_app*), and

the new *db_update* backend service. Each of the parts is built into a self-contained Docker image (the concept is described in the previous chapter in the Section 2.3.1, the implementation details are described in the next chapter in Section 4.2). Running a multi-container application like this requires setting up virtual networks between the containers, passing environmental variables to them and mapping multiple ports inside the containers to ports on the host machine. This can be done by a set of Docker commands with relatively complex commandline arguments. However, this project uses Docker Compose to accomplish this in a much simpler way. Docker Compose allows to save the configuration in a YAML file and run it with one simple command. More information about the tool can be found in its documentation [29], the implementation details are discussed in the Section 4.3.

GitLab can store Docker images in a container registry, which is exclusive for each GitLab project. Each build job then builds the appropriate Dockerfile locally and pushes the image into the project's container registry. Because the deployment jobs are also executed on the shared GitLab runners, they cannot simply start the application. Luckily Docker and Docker Compose allow the commands to be executed on a remote host, which also has Docker and Docker Compose installed, using SSH. By leveraging this feature, the source code of the application, nor the GitLab variables never make it to the actual server hosting the application. They are loaded only into the runner environment and are discarded when the deployment job finishes. The diagram in the Figure 3.8 shows this in more detail.

The deployment can be done in two environments - production and stage. The application can be deployed to both environments at once. The environments do not share the graph database contents and they run on the same server, but on different ports. The stage environment can be used to test out unreleased changes. Deployment to the production environment is done automatically from the git master branch. Deployment to the stage environment can be triggered manually on any git branch other than master.

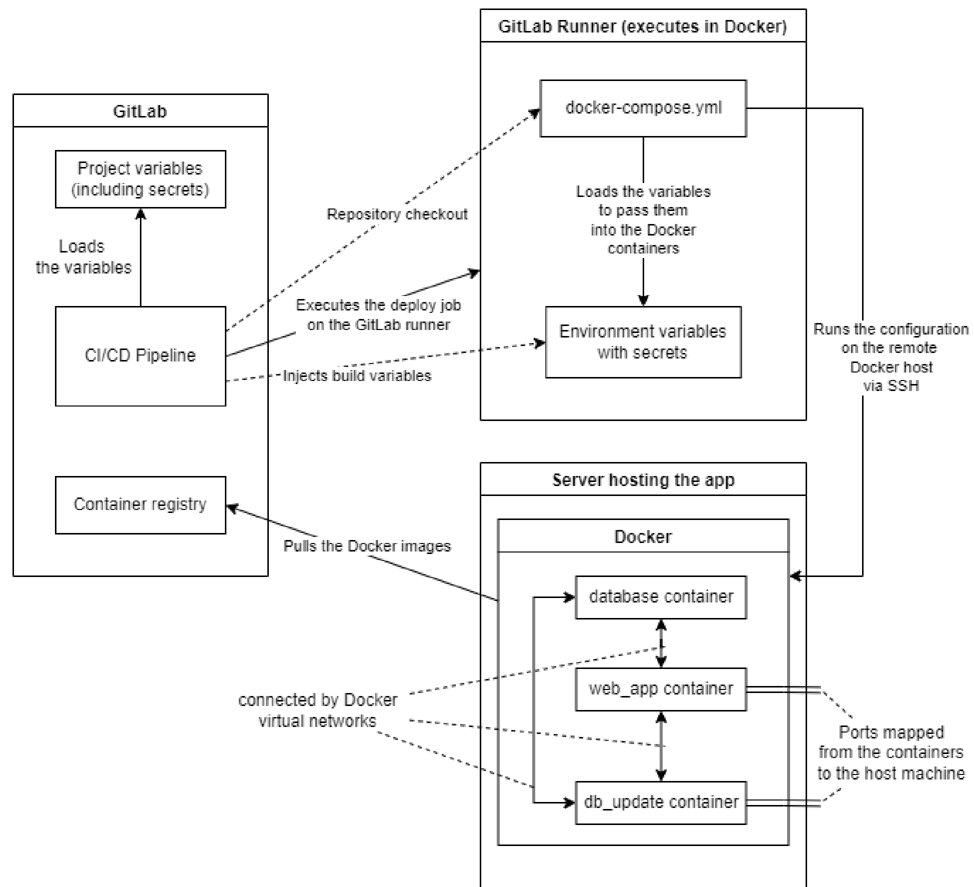


Figure 3.8: Diagram of the GitLab deployment design

Chapter 4

Implementation

After designing the application in the previous chapter, it is time to describe some of the implementation in more detail. As described in the Section 2.2.3, the original application contains the *db_update* module which provides some static information about the collections present in the database. This chapter describes the Python packages in the new *db_update* backend application, described in the Section 3.2, the implementation of the Dockerfiles used to dockerize each part of the application and finally the implementation of the GitLab CI/CD pipeline. Unless specified otherwise, the following sections will refer to the *db_update* backend service simply as *application*. The section contains a bunch of Python code snippets to demonstrate the implementation details. To ensure brevity, the snippets often leave out some parts of the code, like for example the imports, that are not directly related to the purpose of the snippet.

4.1 Application structure

The whole application is a runnable Python module. This means, that the root package of the application contains a file named *__main__.py*. This file acts as an entrypoint for the module. The module can be run from the commandline with `python -m db_update`. This entrypoint is intended only for running the application in a development environment for debugging purposes. The way to run the application in production is via an ASGI [30] server, like Uvicorn [31]. This is reflected in the application's Dockerfile.

The application is comprised of seven top-level packages. Some of them contain one or more modules and some of them contain other nested packages as well. The following subsections will go into more details on each of them. The main packages are the following:

- config
- database
- exception
- model
- routers

- services
- utils

4.1.1 Config

The *config* package contains two modules. The first module, *constants.py*, holds some constant values used throughout the application. The other module, *config.py*, holds all the configurable variables for the rest of the application. The variables are divided into logical groups. Each of these groups is implemented as a dataclass from the *pydantic* [32] library, which uses type hints to validate the attribute types and raises user friendly errors when the validation fails. An example of such dataclass can be seen in the code snippet in Figure 4.1.

```
@dataclass(frozen=True)
class DbConnection:
    host: str
    port: int
    username: str
    password: str
    database: str
```

Figure 4.1: Example of a class holding a group of configuration variables

These groups are instantiated as class attributes of a main Config class with values loaded from the system environment. This class can be seen in the code snippet in the Figure 4.2. The `Config.load_env()` function is called during application initialization. The rest of the application can then import the *Config* class and access its attributes to get the configuration values in the following way: *Config.db_connection.password*.

4.1.2 Database

The *database* package is made of two modules - *database_connection.py* and *db_api.py*. The *database_connection* module establishes connection with the Graph database using the SQLAlchemy engine and the configuration values. It also provides a SQLAlchemy session maker object to the rest of the application. A session object is needed to execute queries against the database. The *db_api* module encapsulates the creation and execution of SQL queries and provides them as Python functions to the rest of the application. As can be seen in the code snippet in the Figure 4.3, the function uses the SQL building abstraction mentioned in the previous chapter in the Section 3.4.1.

```
class Config:
    @classmethod
    def load_env(cls) -> None:
        try:
            cls.db_connection = DbConnection(
                os.getenv("DB_HOST"),
                os.getenv("DB_PORT"),
                os.getenv("DB_USR"),
                os.getenv("DB_PSW"),
                os.getenv("DATABASE"),
            )
            cls.path = ...
            # rest of the groups omitted for brevity
        except ValueError as err:
            raise ConfigLoadError(
                f"Error while loading configuration\n{err}"
            )
```

Figure 4.2: The main Config class holding all the configuration groups

```
async def select_uncomputed_graphs(
    session: Sess, ignored_g6: list[str], limit: int
) -> Iterable[Graph]:
    """Select a number of graphs
    where the `computed` column is `False`.

    :param session: Database session
    :param ignored_g6: Graphs to exclude
    :param limit: The number of graphs to return
    :return: Iterable of the found graphs
    """
    stmt = select(Graph).where(Graph.computed == False)
    stmt = stmt.where(Graph.g6.not_in(ignored_g6))
    stmt = stmt.limit(limit)
    return (await session.execute(stmt)).scalars()
```

Figure 4.3: An example of a Python function encapsulating a database query

4.1.3 Exception

This package contains two modules - *exceptions.py* and *handlers.py*. The *exceptions* module defines some custom exceptions that can be raised throughout the application. The *handlers* module contains functions that are registered in FastAPI to handle specific exceptions raised during request processing. An

example of such handler can be seen in the code snippet in the Figure 4.4. This handler is called whenever a database *IntegrityError* is raised. That happens when a unique constraint would be violated by an SQL query. In case the exception is raised, the handler responds with the appropriate HTTP code and message automatically. Thanks to this, the exception does not need to be caught in the code that's attempting to run the potentially dangerous query. It prevents code duplication of needing to set up a try catch block and responding accordingly. It allows for an overall cleaner code in the rest of the application.

```
async def db_conflict(
    _, exc: IntegrityError
) -> tuple[Message, int]:
    match = re.match(
        r".*DETAIL:\s+[\w()\s]+=\((.*)\)",
        exc.orig.args[0],
        re.DOTALL
    )
    column_val = match.group(1).strip()
    return JSONResponse(
        status_code=status.HTTP_409_CONFLICT,
        content=Message(
            detail=f"{column_val!r} already exists"
        ).dict(),
    )
```

Figure 4.4: An example of a Python function used as an exception handler

4.1.4 Model

The *model* package contains an *api* package with classes defining the API request and response payloads. Similarly to the configuration classes, they use the *pydantic* library and type hints to validate attribute types. FastAPI also uses them to generate the documentation. Apart from that, the model package contains a *dbschema.py* module, which implements the ORM definitions of the database tables. The property columns of the graph table are added dynamically based on an enum implemented in the other module in this package - *graph_properties.py*. As can be seen in the Figure 4.5, the enum in *graph_properties* includes the property name as the enum key and the database column type and and its compute function as the enum value. The compute functions are taken from the original Sage compute engine implementation. In the original application, adding a new graph property required a change in three separate places, as described in the Section 2.2.3. Now, in the extended application, all of that is done by a single modification of this enum class.

```
@dataclass
class PropMeta:
    db_type: TypeEngine
    compute_fn: Callable

class Prop(Enum):
    nodes = PropMeta(Integer, lambda g: g.order())
    edges = PropMeta(Integer, lambda g: g.size())
    components = ...
    # rest of the properties omitted for brevity
```

Figure 4.5: An enum class defining the computable properties of graphs

4.1.5 Routers

The package contains the definitions of the API endpoints. The functions which handle the endpoints are registered using a FastAPI decorator, as can be seen in the code snippet in the Figure 4.6. The decorator takes some keyword arguments which further define its properties. These are used to validate and convert data, and for generating the OpenAPI documentation. The routing functions define the endpoints, but do not implement the application logic. They call functions from the *services* package. That is where the logic is implemented.

The routers package also contains an *auth* package with an *admin.py* module in it. This module implements the administrator rights validation. The parsing and signature validation of the JWT, as described in the Section 3.1.1, is done by the FastAPI framework and the fastapi_oidc library [33]. The *admin* module extends the basic validation by checking the user's roles within the application. Dependency injection provided by the FastAPI framework is then used to secure the administrator endpoints using this check. This can be seen in the code snippet in the Figure 4.7.

4.1.6 Services

The *services* package contains six service modules, each implementing a specific set of features of the application. The services communicate either directly, by one service calling a function from a different service, or indirectly, by one service attaching information to a database session, which is picked up by the other service by its database event hook.

Insert service

The *insert* service contains three functions. The first function is used for adding graphs into the database. It takes a list of canonical Graph6 strings and an optional string *collection* as arguments. If the argument is specified, the graphs are inserted with the collection tag right away. The graphs already

```
router = APIRouter(prefix="/collection")

@router.get(
    "",
    status_code=status.HTTP_200_OK,
    response_model=list[Collection],
    responses={
        status.HTTP_422_UNPROCESSABLE_ENTITY: {
            "model": Message,
            "description": "Incorrect state query",
        },
    },
)
async def get(state: list[CollectionState] = Query(None)):
    """Get collections with an optional `state` filter"""
    collections = await collection_svc. \
                    get_collections(state)
    return [
        Collection(name=c.name, url=c.url, state=c.state)
        for c in collections
    ]
```

Figure 4.6: An example of a function serving as an API endpoint handler

```
from db_update.routers.auth.admin import is_admin

@router.delete(
    "/{name}",
    # decorator keyword arguments omitted for brevity
)
async def delete(name: str, _: IDToken = Depends(is_admin)):
    await collection_svc.remove_collection(name)
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Figure 4.7: An example of a router function using FastAPI dependency injection for authentication

present in the database have the collection tag added by using the upsert method described in the Section 3.1. The Postgres database driver used to perform database queries has a limit of 32767 arguments that can be passed into each query. Because of this, the service splits the list of graphs into multiple database queries and commits the transactions when they are all finished. It also marks the database session with the information that new graphs have been added.

The other two functions are related to the graph adding function and should precede it. The first one is for validating a Graph6 string. It takes in a string and checks whether it is a valid Graph6 format. If it is not, the function raises an exception. The other one is for canonically labeling a list of graphs represented in Graph6 or Sparse6 format using Nauty. Nauty is a program for computing automorphism groups of graphs, created by Brendan McKay and and Adolfo Piperno [34]. It can also produce canonical labels of graphs and has been chosen for this functionality in the original application. The reasoning can be found in the Chapter 3 of Ullrich Herbert's bachelor thesis [4].

Update service

The *update* service acts mainly as a long running task, which ensures the property computation of newly inserted graphs. It maintains a process pool for computing multiple graphs in parallel. The computation itself is done using the *compute* service, which is described in the following Section 4.1.6. The computed results are saved in memory. The service does periodic checks for the number of computed graphs. When there is enough (a configurable variable) of them, it makes a batch update in the database. The batch size should be configured based on the system the application runs on. From the database performance point of view, it is better to make one larger query, than multiple small ones, because each query has the overhead of parsing and planning the query. Larger batch size also permits the periodic checks to be done less frequently. However, storing many computed results while waiting for large batch size consumes more memory. The same limitation for the number of query parameters, as described in the previous section, applies here, making the maximum batch size S_b with the number of updated properties $|P|$ equal to $S_b = 32767 \div |P|$. This constraint is forced when loading the configuration variables. Apart from consuming the computed results, the periodic check also queries the database for uncomputed graphs and schedules their computation.

Compute service

The service provides a function to compute the properties of a given graph using the functions from the enum described in the Section 4.1.4. As can be seen in the code snippet in the Figure 4.8, a new process is spawned for each computation. This is necessary to implement the timeout mechanism, because some of the Sage functions do not respond to alarm nor interrupt signals and must be killed. The function takes a dictionary as an argument. The dictionary represents the graph and optionally its properties. If the property in the dictionary is not None, it is not recomputed again. This allows for easy addition of computable properties in comparison to the original application, described in the Section 2.2.3. When adding a new computable property, the existing graphs only need to be marked as uncomputed during a database

migration (migrations are described in the Section 3.5). The *update* and *compute* services then take care of the rest automatically.

```
def compute_properties(graph: dict) -> dict:
    g6 = graph["g6"]
    g = Graph(
        g6, loops=False, multiedges=False, immutable=False
    )
    result = {"g6": g6}
    parent_end, child_end = multiprocessing.Pipe(False)

    for prop in Prop:
        if graph.get(prop.name, None) is not None:
            continue
        p = multiprocessing.Process(
            target=_wrapper,
            args=(prop.name, g, child_end)
        )
        p.start()
        p.join(Config.timeout.compute_prop)
        if p.is_alive():
            p.kill()
        if parent_end.poll():
            res = parent_end.recv()
            result[prop.name] = _pythonify(res)
        else:
            result[prop.name] = None
            logger.info("%s of %s timed out", prop.name, g)

    return result

def _wrapper(
    prop_name: str,
    graph: Graph,
    pipe: mc.Connection
) -> None:
    result = Prop[prop_name].value.compute_fn(graph)
    pipe.send(result)
```

Figure 4.8: The implementation of property computation with timeout

Download service

The *download* service implements downloading of the three typical collection file formats explored in the Section 2.1.3. Even though this is a very

straightforward task, there are some things, that can break the whole application, if done incorrectly. Downloading and decompressing large files can take a lot of time. If not scheduled correctly using the *await* keyword, as described in the Section 3.6.4, it could block the whole application from processing other incoming requests. Downloading content over network is a perfect example of how *asyncio* and its *await* should be used. Therefore, as can be seen in the code snippet in the Figure 4.9, an *asyncio* compliant library for HTTP requests called *httpx* is used to download the content of the collection URL. The decompression is delegated to a different thread using the *asyncio.to_thread* method and *awaited* as well. The method returns a list of bytes, each representing a graph in a yet uknown format. Bytes are used even though all the formats currently supported by the extended application are text formats. This makes the system easily extensible by binary graph formats, that might be desired in the future.

Convert service

The *convert* service is used to follow up on the graphs downloaded by the *download* service. Currently the service can convert the three formats mentioned in the Section 2.1.3. Internally, there is a convert function for each supported graph format, that takes in a bytes object, attempts to parse it assuming it is in the given format and returns a Graph6 string. If the parsing fails, it is evaluated that the given input does not have the format expected by the function and an exception is raised. The service provides a method that takes in a list of bytes (as returned by the *download* service), and tries to convert each item using the above-mentioned convert functions until one of them successfuly converts the whole list into Graph6 strings. If none of the convert functions succeeds, an exception is raised. This approach makes the service easily extensible of other formats should the need arise in the future. All that is needed is to implement a function with the same interface (input bytes object, return Graph6 string, raise an exception if the input format is different). Thanks to the input being bytes and not a string, even binary graph formats can be included easily.

The implementation of two of the convert function can be seen in the code snippet in the Figure 4.10. They both use the fact, that the Graph class provided by Sage can be instantiated by a Sparse6 string or a list of edges, and is able to output the Graph6 string of the loaded graph.

Collection service

This service is the most complex of all. It provides some basic logic behind the collection API endpoints. As can be seen in the code snippet in the Figure 4.11, these functions consist of only one database query. This very simple implementation is possible mainly thanks to the exception handlers described in the Section 4.1.3. However, the service contains more complex methods as well. For example the function responsible for importing a collection directly calls other services to download, convert and insert the

```
import httpx

async def get_graphs(url: str) -> list[bytes]:
    try:
        async with httpx.AsyncClient() as client:
            content = (await client.get(
                url, follow_redirects=True
            )).content
        filename = url[url.rfind("/") + 1 :]
        if filename.endswith(".gz"):
            res = await asyncio.to_thread(
                gzip.decompress, content
            )
            return res.splitlines()
        bytes_io_content = io.BytesIO(content)
        if zipfile.is_zipfile(bytes_io_content):
            tmp_dir = Config.path.tmp_dir / str(uuid4())
            await asyncio.to_thread(tmp_dir.mkdir)
            try:
                zip_f = zipfile.ZipFile(bytes_io_content)
                res = await asyncio.to_thread(
                    zip_f.extractall, tmp_dir
                )
                return [
                    f.read_bytes().strip()
                    for f in tmp_dir.glob("*")
                ]
            finally:
                await asyncio.to_thread(
                    shutil.rmtree, tmp_dir
                )
        return content.splitlines()
    except Exception as e:
        raise InvalidCollection
```

Figure 4.9: The implementation of the *get_graphs* function from the *download* service

graphs from the collection. The service also implements a database commit hook, which checks whether any collection in the approved state should be moved to the imported state, when all its graphs are successfully computed by the *update* service.

```python
from sage.all import Graph

def _sparse6(graph: bytes) -> str:
    return Graph(
        graph.decode(),
        format="sparse6",
        loops=False,
        multiedges=False,
        immutable=True,
    ).graph6_string()

def _edge_list(graph: bytes) -> str:
    def parse_edge(edge: str) -> tuple[int, int]:
        delim_idx = edge.find(" ")
        return (
            int(edge[:delim_idx]),
            int(edge[delim_idx + 1 :])
        )

    edges = graph.decode().split("  ")
    edge_list = [parse_edge(e) for e in edges]
    return Graph(
        edge_list,
        format="list_of_edges",
        immutable=True
    ).graph6_string()
```

Figure 4.10: Example implementation of two convert functions from the *convert* service

4.1.7 Utils

The *utils* package contains modules with utility functions used in the rest of the application. The most notable one is the *db_session.py* module. To execute database queries using SQLAlchemy, a session must be opened first. When the application needs to do multiple queries in order to achieve something, one session should be used for all the related queries. This way, if something fails before the whole application logic is fullfilled, the session can be discarded without actually affecting the database. On the other hand, when the application logic finishes successfully, the session is commited and all the queries are persisted at once. Opening up a session in each function, that requires database access, would be a lot of code duplication. Moreover, a problem arises, when a function, with an already opened session, calls another function, that would open a different session on its own. Ideally, the already opened session should be passed down to the called function. Both of these

```
@with_session(commit=True)
async def submit_collection(
    name: str, url: str, *, session
) -> None:
    await db_api.insert_collection(session, name, url)
    logger.info("new collection '%s' submitted", name)


@with_session(commit=False)
async def get_collections(
    state: list[CollectionState], *, session
) -> list[Collection]:
    return list(
        await db_api.select_collections(session, state)
    )
```

Figure 4.11: Example implementation of two basic functions from the *collection* service

problems are elegantly solved by a decorator implemented in the *db_session* module.

The use of the decorator can be seen in the code snippet in the Figure 4.11. Each function, that is annotated with the *@with_session* decorator, must be declared with a *session* parameter. When the function is called, the decorator checks, whether the function was called with the session argument and if so, it only passes the session to it. If the session argument is not present, the decorator sets up a session, passes it to the function and potentially commits the session after. This allows for the function to be called in two ways. The examples can be seen in the code snippet in the Figure 4.12.

```
@with_session(commit=True)
def example_function(arg1, arg2, *, session):
    # use the session without caring where it comes from

# 1) the decorator manages the session automatically
example_function(1, 2)
# 2) already existing session is passed as an argument
s: Session = ...
example_function(2, 3, session=s)
```

Figure 4.12: Example of the two possible ways to call a function annotated with *@with_session*

4.2 Dockerization

As described in the Section 2.3.1, a Dockerfile is created to dockerize the applications. The whole project consists of three parts, that are built and deployed separately. This means, that each of them has its own Dockerfile. The following sections will refer to these parts as *services* and they should not be mistaken for the *service* modules described in the previous section. They are the following:

- database
- web application (JavaScript frontend + backend)
- db_update (the new backend service of the extended application)

Dockerfile is built into a docker image. The resulting docker image should be as small as possible and contain nothing that is unnecessary to run the application, like for example the build dependencies. The created Dockerfiles use a multi-stage build to achieve this. Multi-stage builds are divided into stages, where each stage runs in a separate container. This allows to selectively copy artifacts from one stage to another, while throwing away everything else from the intermediate containers. Each stage is based on a predefined image, which is selected using the FROM keyword. The implementation of the Dockerfile for the new *db_update* service can be seen in the code snippet in the Figure 4.13.

The lines 1 to 18 describe the first stage of the build. This stage downloads the source code of Nauty and prepares it for installation on the final image using curl and tar (lines 7-10). It copies the source code of the application into the container (lines 13-15) and packages the application into a built distribution (lines 16-17), that can be installed using Python package manager in the final image.

The lines 20 to 42 describe the second stage. The image of the second stage is based on the *sagemath/sagemath:9.4* image, which has Python and Sage preinstalled. It copies the build artifacts from the previous stage (lines 24, 25) and the migration scripts (line 26). It installs the built distribution of the application and clears the distribution files (lines 28 - 30). Then it compiles and installs the prepared Nauty sources (lines 32 - 35). That is where the build ends. The last ENTRYPOINT command specifies what happens when the container is started. It first runs the database migration tool to ensure the database schema is up-to-date. Finally it starts the application using the Uvicorn [31] webserver.

A similar Dockerfile is used for the dockerization of the JavaScript web application. The first stage of the multi-stage build bundles the frontend code while the second stage gets the bundled files and runs the Express backend serving them.

```
### BUILD ###
# build on the same system as it runs on
FROM python:3.9-slim-buster AS BUILD
# install dependencies
RUN apt update && apt install -y ca-certificates curl
# download nauty
WORKDIR /nauty
RUN curl https://pallini.di.uniroma1.it/nauty27r3.tar.gz \
    -o nauty27r3.tar.gz
RUN tar xvzf nauty27r3.tar.gz
# build the app as a python wheel
WORKDIR /build
COPY setup.py ./setup.py
COPY db_update ./db_update
COPY requirements.txt ./requirements.txt
RUN pip install --upgrade pip build
RUN python -m build --wheel --outdir /dist ./

### RUN ###
FROM sagemath/sagemath:9.4
ENV PATH="/home/sage/sage/local/bin:$PATH"
WORKDIR ./app
# copy build artifacts
COPY --from=BUILD /nauty/nauty27r3 ./nauty
COPY --from=BUILD /dist ./app_wheels
COPY db_migration ./db_migration
# install the app
RUN pip install --upgrade pip setuptools
RUN pip install ./app_wheels/*
RUN sudo rm -rf ./app_wheels
# install wait-for-it and downloaded nauty
RUN sudo apt update && sudo apt install wait-for-it make
RUN cd ./nauty && sudo ./configure && sudo make && \
    sudo make install
RUN sudo rm -rf ./nauty
# run the app
ENTRYPOINT wait-for-it $DB_HOST:$DB_PORT && \
            cd ./db_migration && \
            python3 ./migrate.py upgrade && \
            cd .. && \
            uvicorn --factory "db_update:asgi" \
            --host 0.0.0.0 --port 9000
```

Figure 4.13: Implementation of the *db_update* Dockerfile

4.3 Docker Compose

The containers defined by the Dockerfiles are orchestrated together using Docker Compose, which handles port mapping and passing environment variables to the containers. This configuration is saved in a *docker-compose.yml* file in the root of the project. The compose file defines a set of services, where each service corresponds to one of the containers and its configuration. One of the service definitions can be seen in the code snippet in the Figure 4.14. As can be seen, the compose configuration uses environment variables. The variables used to configure the application, like the database credentials, are injected to the environment from the GitLab variable storage, as described in the Section 3.9. The other variables needed only to correctly execute the compose file, like the deploy environment and image prefix and tag, are provided by the GitLab Pipeline, as described in the following Section 4.4.

```
db_update:
  container_name: db_update_$DEPLOY_ENV
  image: ${IMAGE_PREFIX}db_update${IMAGE_TAG}
  build: ./db_update
  environment:
    DB_HOST:
    DB_PORT:
    DB_USR:
    DB_PSW:
    DATABASE:
  env_file:
    - ${ENV_VARS_DB_UPDATE:-.env}
  networks:
    - graphs_network
  ports:
    - "$DB_UPDATE_PORT:9000"
  depends_on:
    - graphs_db
```

Figure 4.14: Example of a service defined inside a docker-compose.yml file

4.4 GitLab Pipeline

The CI/CD pipeline is implemented in a YAML file called *.gitlab-ci.yml*, stored in the root of the repository. The three separate services can be deployed to two different environments, as described at the end of the Section 3.9. This means, that there are six build jobs and six deploy jobs, which are practically the same. They differ only in the following properties: which one of the three services is the job target and which ports to use based on

the target environment. Fortunately, GitLab Pipeline configuration allows the use of inheritance between the jobs and the creation of so-called hidden jobs. Hidden jobs are never added to the pipeline and are supposed to be used as templates for reusable configuration (therefore the following text sometimes refers to hidden jobs as templates). This way the actual logic of the build and deployment job can be implemented only once as a hidden job, in a generic way parametrizable by variables. The parametrization variables can be also implemented as hidden jobs. The twelve actual jobs can be then implemented as a combination of the templates using inheritance. The template combinations can be seen in the three-dimensional matrix in the Table 4.1.

Pipeline jobs matrix				
		database	web app	db_update
build impl	prod vars	X	X	X
	stage vars	X	X	X
deploy impl	prod vars	X	X	X
	stage vars	X	X	X

Table 4.1: Matrix showing the pipeline job combinations

The choice of the service is implemented as three hidden jobs. One of them can be seen in the code snippet in the Figure 4.15. This template declares the *APP* variable, which is used by the build and deploy implementation job to select the correct service as its target. It also limits the jobs only to commits, which change the files, that actually require a redeploy of the service. For example changing the code of the UI, does not require building and deploying a new version of the *db_update* backend.

```
.db_update_vars:
  variables:
  APP: db_update
  only:
    changes:
      - db_update/**/*
      - docker-compose.yml
      - .gitlab-ci.yml
```

Figure 4.15: Implementation of the *db_update* variables hidden job

The choice of the target environment is implemented as two hidden jobs. One of them can be seen in the code snippet in the Figure 4.16. The variables define the ports on which the application runs in that environment. It also limits the environment to a specific branch, using the *only* attribute.

Two more variables still need to be provided to the compose file, as mentioned in the Section 4.3. One of them is *IMAGE_PREFIX*. This variable

```
.prod_vars:
  variables:
    DEPLOY_ENV: prod
    DB_UPDATE_PORT: $PORT_PROD_DB_UPDATE
    WEB_APP_PORT: $PORT_PROD_WEB_APP
  only:
    refs:
      - master
```

Figure 4.16: Implementation of the production variables hidden job

is used to identify the GitLab container registry where the built docker images are stored and pulled from during deployment, as described in the Section 3.9. Docker also needs to be authenticated with the registry. The registry image name, url and credentials are automatically set as environmental variables inside every job as *$CI_REGISTRY_IMAGE*, *$CI_REGISTRY* *$CI_REGISTRY_USER* and *$CI_REGISTRY_PASSWORD*. The other variable needed by the compose file is *IMAGE_TAG*, which specifies the image inside that registry. The tags *prod* and *stage* are used. This variable is derived from the variable supplied by the hidden job specifying the target environment. The job also needs to execute docker commands via SSH when deploying the application, as described in the Section 3.9. In order to do that, SSH keys and known hosts need to be set up first. These values are also saved as the GitLab project variables. One more level of inheritance is used to achieve this, as can be seen in the code snippet in the Figure 4.18.

The twelve final jobs are then implemented as jobs, which extend multiple templates, as can be seen in the example in the code snippet in the Figure 4.17.

```
build_prod_db_update:
  extends:
    - .build
    - .prod_vars
    - .db_update_vars
```

Figure 4.17: Job created by a combination of templates

```yaml
.docker_base:
  image: docker:20.10.14
  services:
    - docker:dind
  variables:
    IMAGE_PREFIX: "$CI_REGISTRY_IMAGE/"
    IMAGE_TAG: ":$DEPLOY_ENV"
    DOCKER_DRIVER: overlay2
    DOCKER_TLS_CERTDIR: "/certs"
  before_script:
    - apk add ca-certificates docker-compose
    - docker login -u $CI_REGISTRY_USER \
        -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh
    - echo "$SSH_KNOWN_HOSTS" >> ~/.ssh/known_hosts

.build:
  extends:
    - .docker_base
  stage: build
  allow_failure: false
  script:
    - docker-compose build --no-cache $APP
    - docker-compose push $APP

.deploy:
  extends:
    - .docker_base
  stage: deploy
  when: on_success
  retry: 2
  script:
    - docker-compose -H "ssh://$SERVER_USER@$SERVER_IP" \
        -p "graphs_$DEPLOY_ENV" down $APP
    - docker-compose -H "ssh://$SERVER_USER@$SERVER_IP" \
        -p "graphs_$DEPLOY_ENV" pull $APP
    - docker-compose -H "ssh://$SERVER_USER@$SERVER_IP" \
        -p "graphs_$DEPLOY_ENV" up -d --no-deps $APP
```

Figure 4.18: The implementation of build and deploy hidden jobs

Chapter 5
Conclusion

The goal of this master thesis was to propose and implement an extension of an existing web application called Web Graph Service. The main goal of the extension is to provide unified access to graphs and their properties from various Internet sources. The extension also requires to automate the process of deployment of the current and future modifications of the application.

The core of the extension has been built as a Python backend service. It is able to import graph collections stored in three different file formats containing graphs also in three different graph formats (Graph6, Sparse6, Edge list). The list of supported formats is ready to be easily extended even of non-textual graph formats. Some of the service's API is accessible only by administrators and the authorization is done via CTU single sign-on. The source code is commented using standardized Python docstrings, from which documentation is generated, available at `https://gitlab.fel.cvut.cz/graphs/development/-/tree/master/db_update/docs`. The application is fully dockerized and has an automated deployment pipeline. This will make life much easier for the future developers of the application.

The CTU single sign-on authorization is convenient, but it makes it impossible to grant administrator rights to anyone outside of the CTU domain. This is unlikely to become an issue. However, should the need for external administrator arise, the authorization would need to be completely rewritten.

5.1 Future improvements

The new service in the extended application has been built as a monolith, which serves the database management API as well as computes the graph properties using a process pool. This allows for a parallelization of computing multiple graphs at once. However, it is limited by the number CPU cores of the machine on which the service runs. Greater parallelization could be achieved by splitting the service into multiple microservices and creating an external queue for graph computations. The compute microservice would run as a single process consuming the queue entries one by one. This implementation would provide greater vertical scalability, as the number of consumers could be increased or decreased based on the queue size. They could also run on different machines or potentially in a cloud, utilizing many CPU cores.

The current way of storing the graphs and ensuring there are no isomorphisms relies on the unique constraint of the Graph6 column in the database. However, as metioned in the Section 3.1, this implies a limit of 2730 bytes on the Graph6 representation of each graph. The limit cannot be simply converted to a number of graph vertices, because Postgres compresses the data before storing and indexing them. However, by testing the limits it was found, that graphs above 600 vertices start getting rejected because of this. The standard approach to overcome this limit would require adding a column containing a hash of the Graph6 string and setting up the unique constraint on that, instead of the Graph6 column itself. However, this comes with a risk of running into hash collisions.

Bibliography

[1] Diestel, R., 2000. *Graph theory*. New York: Springer, p.3.

[2] Hog.grinvin.org. 2022. *House of Graphs*. [online] Available at: `https://hog.grinvin.org/` [Accessed 26 April 2022].

[3] McKay, B., 2022. *graph formats*. [online] Users.cecs.anu.edu.au. Available at: `https://users.cecs.anu.edu.au/~bdm/data/formats.html` [Accessed 1 May 2022].

[4] Ullrich Herbert. *User Extensible Graph Database*. Czech Technical University in Prague, 2018. [CTU Bachelor thesis]

[5] McKay, B., 2022. *Brendan McKay's Home Page*. [online] Users.cecs.anu.edu.au. Available at: `http://users.cecs.anu.edu.au/~bdm/` [Accessed 27 April 2022].

[6] Atlas.gregas.eu. 2022. *GReGAS Atlas*. [online] Available at: `http://atlas.gregas.eu/` [Accessed 2 May 2022].

[7] Expressjs.com. 2022. *Express - Node.js web application framework*. [online] Available at: `https://expressjs.com/` [Accessed 1 May 2022].

[8] Graphviz - Graph Visualization Software. (n.d.). *Graphviz.org*. [online] Available at: `https://graphviz.org/` [Accessed 27 April 2022].

[9] Reactjs.org. 2022. *React – A JavaScript library for building user interfaces*. [online] Available at: `https://reactjs.org/` [Accessed 1 May 2022].

[10] webpack. 2022. *webpack*. [online] Available at: `https://webpack.js.org/` [Accessed 1 May 2022].

[11] SageMath Mathematical Software System. 2022. *SageMath Mathematical Software System - Sage*. [online] Available at: `https://www.sagemath.org/` [Accessed 1 May 2022].

[12] Networkx.org. 2022. *NetworkX — NetworkX documentation*. [online] Available at:
https://networkx.org/ [Accessed 1 May 2022].

[13] Wolfram.com. 2022. *Wolfram: Computation Meets Knowledge*. [online] Available at:
https://www.wolfram.com/ [Accessed 1 May 2022].

[14] PostgreSQL Documentation. 2022. *psql*. [online] Available at:
https://www.postgresql.org/docs/current/app-psql.html [Accessed 1 May 2022].

[15] CTU - Faculty of Electrical Engineering. 2022. *IT Centre (SVTI)*. [online] Available at:
https://svti.fel.cvut.cz/en/ [Accessed 3 May 2022].

[16] PostgreSQL Documentation. 2022. *pg_dump*. [online] Available at:
https://www.postgresql.org/docs/current/app-pgdump.html [Accessed 1 May 2022].

[17] Docker. 2022. *Home - Docker*. [online] Available at:
https://www.docker.com/ [Accessed 1 May 2022].

[18] Tomáš Roun. *Graph Database Fundamental Services*. Czech Technical University in Prague, 2018. [CTU Bachelor thesis], p.51.

[19] Openid.net. 2022. *Final: OpenID Connect Core 1.0 incorporating errata set 1*. [online] Available at:
https://openid.net/specs/openid-connect-core-1_0.html [Accessed 8 May 2022].

[20] Sqlalchemy.org. 2022. *Features - SQLAlchemy*. [online] Available at:
https://www.sqlalchemy.org/features.html [Accessed 8 May 2022].

[21] Alembic.sqlalchemy.org. 2022. *Auto Generating Migrations — Alembic 1.7.7 documentation*. [online] Available at:
https://alembic.sqlalchemy.org/en/latest/autogenerate.html
[Accessed 8 May 2022].

[22] Djangoproject.com. 2022. *The web framework for perfectionists with deadlines | Django*. [online] Available at:
https://www.djangoproject.com/ [Accessed 3 May 2022].

[23] Flask.palletsprojects.com. 2022. *Welcome to Flask — Flask Documentation (2.1.x)*. [online] Available at:
https://flask.palletsprojects.com/en/2.1.x/ [Accessed 3 May 2022].

[24] Ivory.idyll.org. 2022. *An Introduction to the Python Web Server Gateway Interface (WSGI)*. [online] Available at:
http://ivory.idyll.org/articles/wsgi-intro/what-is-wsgi.html
[Accessed 6 May 2022].

[25] Connexion.readthedocs.io. 2022. *Welcome to Connexion's documentation! — Connexion 2020.0.dev1 documentation.* [online] Available at: `https://connexion.readthedocs.io/en/latest/` [Accessed 3 May 2022].

[26] Swagger.io. 2015. *API Resources.* [online] Available at: `https://swagger.io/resources/open-api/` [Accessed 3 May 2022].

[27] Fastapi.tiangolo.com. 2022. *FastAPI.* [online] Available at: `https://fastapi.tiangolo.com/` [Accessed 6 May 2022].

[28] Techempower.com. 2022. *TechEmpower Framework Benchmarks.* [online] Available at: `https://www.techempower.com/benchmarks/#section=test&runid=7464e520-0dc2-473d-bd34-dbdfd7e85911&hw=ph&test=composite&l=v2qiv3-db&a=2&f=4ftlb4-1-5slc-0-4zt38-18y68-4-18y80-w-18y6k-1w4qp-0` [Accessed 6 May 2022].

[29] Docker Documentation. 2022. *Overview of Docker Compose.* [online] Available at: `https://docs.docker.com/compose/` [Accessed 8 May 2022].

[30] Asgi.readthedocs.io. 2022. *Introduction — ASGI 3.0 documentation.* [online] Available at: `https://asgi.readthedocs.io/en/latest/introduction.html` [Accessed 9 May 2022].

[31] Uvicorn.org. 2022. *Uvicorn.* [online] Available at: `https://www.uvicorn.org/` [Accessed 9 May 2022].

[32] Pydantic-docs.helpmanual.io. 2022. *pydantic.* [online] Available at: `https://pydantic-docs.helpmanual.io/` [Accessed 9 May 2022].

[33] Winters, H., 2022. *GitHub - HarryMWinters/fastapi-oidc.* [online] GitHub. Available at: `https://github.com/HarryMWinters/fastapi-oidc` [Accessed 12 May 2022].

[34] McKay, B.D. and Piperno, A., 2014. *Practical Graph Isomorphism, II.* Journal of Symbolic Computation, 60, p. 94-112

Appendix A
Graph formats

This is a list of the descriptions of the widely used graph formats. The formats Digraph6, GraphML and DOT are not relevant for this thesis, but are included for the sake of completeness.

- Graph6, Sparse6, Digraph6
  `http://users.cecs.anu.edu.au/~bdm/data/formats.txt`
- Edge list
  `https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs`
- GraphML
  `http://graphml.graphdrawing.org/primer/graphml-primer.html`
- DOT
  `https://graphviz.org/doc/info/lang.html`

Appendix B
Source code

The source code can be found in the attachments in *src.zip* or in the git repository at `https://gitlab.fel.cvut.cz/graphs/development`.