MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Backend patient API for AICope project

Bachelor's Thesis

MATEJ DIPČÁR

Brno, Fall 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Backend patient API for AICope project

Bachelor's Thesis

MATEJ DIPČÁR

Advisor: doc. Mgr. Bc. Vít Nováček, PhD

Department of Machine Learning and Data Processing

Brno, Fall 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Matej Dipčár

Advisor: doc. Mgr. Bc. Vít Nováček, PhD

Acknowledgements

This thesis would not have been possible without all the valuable advice of my consultant, Bc. Daniel Plakinger, who had to endure my never-ending questions and comments. Furthermore, my sincere gratitude goes to the advisor, doc. Mgr. Bc. Vít Nováček, PhD, for offering such an interesting topic to work on and for the technical help provided. I would also like to thank my parents and my loving girlfriend, who immensely supported me in my studies and while writing this thesis.

Abstract

This thesis aims to be a valuable contribution to the University's project "AICope" developed in collaboration with the Masaryk Memorial Cancer Institute. The main goal of this project is to develop an application designed for patients diagnosed with breast cancer. The main contribution this thesis brings, but is not limited to, the implementation of two essential backend services, namely Patient API and Gateway API, which were integrated into the project. The implications this project bears are significant within the medical field, as it strives to build a foundation for future development in this direction. The achieved results are discussed at the end of this thesis.

Keywords

graphql, grpc, patient-oriented research, nodejs, nestjs, express, postgresql, typescript, javascript, secure cloud, cerit, helm, terraform, kubernetes, docker

Contents

List of Figures

Introduction

This thesis contributes to the University's project "AICope"[1], under the project code MUNI/G/1763/2020. This project was developed in collaboration with the Masaryk Memorial Cancer Institute (MMCI). Subsequent references to the institute in this thesis will use the abbreviation MMCI.

A patient empowerment application that is the main research outcome of the AICope project was specifically designed for and with the help of patients diagnosed with breast cancer being treated at the MMCI. The psychological state of the patient plays a non-negligible role during the treatment of cancer or any medical condition and is directly affected by the patient's understanding of their medical condition[1]. This project aims to provide patients diagnosed with breast cancer from the MMCI, and possibly patients with other diagnoses as well in the future, a web portal where they can learn about their medical condition. Although there are many ways for a patient to be educated about their medical condition, most of the time, the only personalised way to educate about it is during the visit with the doctor. This project seeks to provide patients with a personalised way to educate themselves about their condition. Furthermore, as the final application should educate the patients, one of the expected results is to reduce the workload of the doctors caring for these patients. This should allow doctors to focus on more critical aspects of patient care rather than spending time educating them.

This thesis is focused on implementing two essential backend services of the final application, as well as many other contributions and adjustments of the other services and components of the application.

1. https://www.fi.muni.cz/app/projects?project=60489

1 Background

Thanks to the collaboration with MMCI, a group of patients who agreed to participate in this research and development were selected. The application was developed in several iterations while constantly consulting the application's state with the patients. As a result, many problems that were not obvious to a person developing the system were discovered and recognised, and many suggestions and features the patients would appreciate in the system were incorporated.

To address the project's most critical goal—delivering personalised information about patient conditions—several aspects had to be considered: how to retrieve, process and store data about patient conditions, and finally, how to provide personalised information.

The source of the data about patients is patient reports—the reports a doctor creates during each patient visit or examination. Each report is exported from the MMCI into an XML file containing some basic information about the patient and all the reports of all of their doctor visits and examinations. During the development of the application, a set of retrospective anonymised reports about previous patients was provided by MMCI to help with the development. Additionally, reports about the selected group of patients were provided at the end of the development. The application was launched for this set of patients to test the application.

To assess the effectiveness of the project, a survey evaluating the level of informativeness of each patient's medical condition was conducted before the test and is also planned to be conducted after the test is completed.

The application was developed using microservice architecture. As the application consists of many services, and not all are relevant to the scope of this thesis, only the relevant services will be considered. Subsequently, the application will be presented in a simplified form by excluding the irrelevant services.

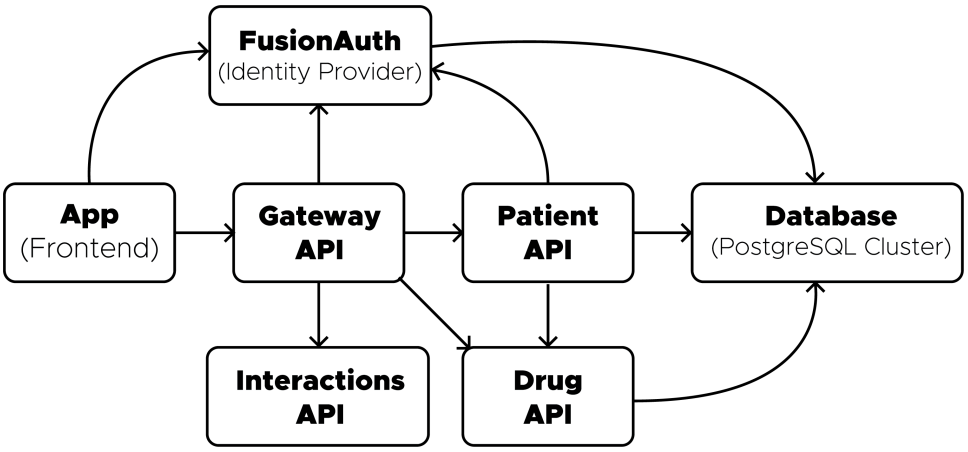

Figure 1.1: Microservice architecture diagram

Figure 1.1 describes the final microservice architecture diagram, all relevant services, and their dependencies. As the objective of this thesis was to implement the Gateway API and Patient API services, they will be the primary focus of the following chapters and sections.

In this section, the above-described services are introduced. The frontend application provides a user interface accessible to both patients and doctors. The Gateway API service acts as an intermediary, enabling communication between the frontend and the rest of the backend services. For user management, FusionAuth, a third-party identity provider was used. Patient API is dedicated to managing patient data and medical reports, whereas the Drug API is responsible for the management of drugs and their allocation to patients. Additionally, the Interactions API offers insights into potential drug interactions. To ensure consistent and reliable data storage across these services, a PostgreSQL database was utilised as the persistent data layer.

To better imagine the whole process, we can consider the following example. A patient logs into the application and clicks a button to view all reports that are associated with this patient. In order for the frontend to get a hold of this data, it sends a request to the Gateway API. The Gateway API service first attempts to authenticate and authorise the user who sent the request, and if it succeeds, continues to retrieve the requested data. However, since the Gateway API does not have

direct access to any data, it has to retrieve them from the individual services. Specifically, to retrieve data about the reports, a request to the Patient API is sent. The Patient API, upon receiving the requests, looks up the data in the database and sends them back to the Gateway API, which will afterwards get sent back to the UI. After the UI receives the data, it attempts to display them to the user. Although this is a very simplified example, it was introduced to create a bigger picture of how these services interact with each other.

To clarify my own contribution to the project, it's important to note that I did not implement the Drug API, the Interactions API, and the frontend application. My involvement was focused on the Gateway API and the Patient API. Additionally, I have contributed to the development of annotators, which will be discussed later, as well as the provisioning of the system.

2 Drug API

Drug API is one of the many services within the system relevant to this thesis. The reason to start the thesis with this service is that understanding the purpose of the service and the API it provides is vital for understanding the rest of the thesis.

Drug API, as the name suggests, is a service providing information about drugs. It is programmed in a Go language[1] and provides its API using the gRPC (Remote Procedure Calls) framework[2].

2.1 gRPC API

An essential aspect of gRPC APIs to consider is the fact that compared to the popular REST APIs, the message format is not JSON but rather a protobuf (protocol buffer)—a binary compressed format. gRPC API is defined in a protobuf `.proto` file. It contains a set of services, each providing a set of available methods. Drug API provides three services: `DrugService`, `PatientService`, and `PatientDrugService`.

2.1.1 DrugService

This service offers 3 API methods:

- **Get** — The `Get` method expects a drug ID (a UUID drug identification) and outputs the requested drug.
- **BatchGet** — The `BatchGet` expects a list of drug IDs and returns a list of requested drugs.
- **List** — The List method is slightly more complex. A pagination was introduced to let the client choose how many drugs to fetch. This is done to prevent fetching abnormally many drugs, which would increase resources used by both services, causing disruptions and possible instability due to resource utilisation. The method takes a page size (a limit of how many drugs to fetch) and a page number (a reference to a page) and provides

1. https://go.dev/
2. https://grpc.io/

a requested number of drugs on a requested page. Moreover, it also accepts a few filtering fields to narrow down the results. For instance, a field containing an ATC code (Anatomical Therapeutic Chemical (ATC) Classification[3] – a more complex universal drug classification).

2.1.2 PatientService and PatientDrugService

Since Drug API not only provides information about drugs but also aims to connect patients with their assigned drugs, these two services exist. `PatientService` provides two methods:

- **Create** — The `Create` method registers a patient in Drug API. It is expected for the patient to already have a user account created, as its ID is expected to be provided in the request.
- **Get** — The `Get` method expects either the user account ID or the patient ID within the Drug API service and returns the patient.

This `PatientDrugService` exists solely to facilitate the existence of `PatientDrugService`, which manages the relationship between the patient and their assigned drugs. It offers these four methods:

- **Create** — The `Create` method expects a drug ID, a patient ID, the timestamp of when the drug was assigned, and the entity that assigned the drug. Afterwards, it assigns a drug to a patient.
- **BatchCreate** — The `BatchCreate` method is quite similar to the `Create` method but instead expects a list of drug IDs and assigns all of them to the patient.
- **Delete** — The `Delete` method deletes a single drug from the patient's assigned drugs. It expects a patient ID, a drug ID, and a flag denoting whether the deletion should be soft or hard. Hard delete deletes the entry entirely, while soft delete only marks the entry as deleted.

3. https://www.whocc.no/atc_ddd_methodology/history/

- **HardDelete** — The `HardDelete` method deletes all drugs from the patient's assigned drugs that match the provided filters. It expects either a patient ID, a `created_by` field, or both. A flag indicating whether the deletion should be soft or hard is also expected.

- **Get** — The `Get` method expects either the patient ID or the entity that assigned drugs. It either outputs a list of all patient's assigned drugs or all drugs the specified entity has assigned.

2.1.3 Metadata

Drug API also requires an API key to be sent in the metatada of each request. The API key is needed to ensure that only services with this API key can communicate with the Drug API service. Furthermore, it also serves to avoid potential conflicts when deploying the service in different environments by preventing the services from accessing each other throughout various environments.

Due to the inter-service communication in the whole system, identifying each request traversing these services is crucial. Consequently, a UUIDv4 request ID is used and propagated among services, enabling the request to be traced back to its source. Drug API requires this request ID in the metadata of each request.

3 Patient API

Patient API is an essential service of the application. As the name suggests, this service takes care of patients and their data. Patients in the application can be viewed as users who can access the application and interact with it. The primary goal of this service is to provide an API for managing these patients—to provide methods to create, modify, retrieve, and search patients in the application.

3.1 Underlying Technologies

3.1.1 Language

First and foremost, the programming language selected for this service is TypeScript, a strongly typed language and a super-set of JavaScript. It was preferred in comparison to JavaScript due to its strong typing, as it can avoid numerous problems by identifying potential runtime errors beforehand. Compared to other programming languages, TypeScript is great for quick prototyping, has a rich ecosystem and is ranked in the top 5 most used programming languages in the year 2023 according to StackOverflow [2].

3.1.2 Runtime

Nowadays, TypeScript can be run in multiple runtimes like NodeJS, Bun or Deno. For this service, NodeJS was utilised along with a transpiler. Due to its robustness, broad community support, and rich package ecosystem, NodeJS stood out as the ideal choice. Despite the advantages Deno offers over NodeJS, the decision was ultimately swayed by the rich ecosystem that comes with using NodeJS. However, if Bun had existed when the project started, it most likely would have been the selected runtime for this service.

3.1.3 Framework and API

Since this service does not directly communicate with the frontend, the choice of the API was not restricted. Therefore, for its simplicity and scalability, a RESTful API was implemented. As for the framework,

the most popular lightweight web server frameworks are Express.js[1] and Fastify[2]. Since Express.js has existed for more than twice as long as Fastify, its ecosystem has naturally developed more over time. Furthermore, the primary benefit of Fastify over Express.js is performance, especially in handling many requests per second [3]. However, considering our application is not expected to grow as much, the benefits of Express.js outweighed the ones of Fastify.

3.1.4 Database

Due to the service's need to store data about patients and their corresponding reports, a database had to be incorporated. Considering that many other services within the system also use a database, the existing PostgreSQL database was used in order to maintain a simpler and more uniform architecture. For communication with this database, an NPM package `pg`[3] was used. Although, in many cases, using an ORM rather than a simple client can be beneficial, given that this service is supposed to be relatively small, the additional overhead of an ORM layer appears to exceed the benefits it might offer.

3.1.5 Identity and Access Management

The whole system will need to handle many users in a secure manner, which suggests some service for user management is highly preferred. KeyCloak[4] and FusionAuth[5] were the two IAMs (Identity and Access Management systems) that had to be considered. I have found FusionAuth relatively easy to set up, use, and administrate, whereas KeyCloak was slightly more complex. On the other hand, KeyCloak is an entirely free open-source option, whilst FusionAuth is free of charge only up to a certain extent—a limit of a maximum number of users that can be managed under the free version. Despite its pricing model, I still found FusionAuth as a better option.

1. https://www.npmjs.com/package/express
2. https://www.npmjs.com/package/fastify
3. https://www.npmjs.com/package/pg
4. https://www.keycloak.org/
5. https://www.fusionauth.io/

In conclusion, the choice to use an IAM proved to be the right decision, as it helped the project delegate most of the security concerns to the FusionAuth and, therefore, focus on the more essential aspects of the project.

3.1.6 Workflow Management

One of the crucial goals of the application was to display the patient reports written by doctors. Given the technical nature of these reports, it was essential to transform them into a format understandable to individuals without expertise in the field, a process necessary to support the educational aspect of the application. As a result, part of our team was working on several annotators that would take a patient report as input, recognise as many relevant keywords as possible, and annotate them with XML tags. Thus, each of these keywords, which could be the name of a drug, a diagnosis, or generally anything that could be seen as technical, was annotated, and a context describing this term was provided in the annotation. For instance, a drug annotator annotates all drugs in the report and includes information about them, such as their ID.

Near its end, the project reached a point where several scripts, each annotating different terms of the reports or simply performing some transformations, had to be executed. Since managing and running a complex workflow such as this is cumbersome, a workflow management tool was incorporated. Apache Airflow appeared to be the tool for this job as it offered an intuitive way for defining and connecting tasks, a wide range of operators for running these tasks, and a valuable UI for monitoring and scheduling jobs. Further details about the choice of this tool and its analysis in the context of the AICope project are discussed in David Rusnák's thesis, "DevOps and MLOps Pipelines in a Secure Kubernetes Cluster", particularly in the Apache Airflow section [4].

3.2 Implementation

3.2.1 Configuration

Considering that the configuration is relatively simple, mainly consisting of key-value pairs, a more advanced configuration was unnecessary; thus, a `.env` file was sufficient. It is a simple text file containing key-value pairs that are loaded into environment variables upon starting the service. This approach is commonly used for passing configurations or sensitive information like keys rather than including them directly in the code. The dotenv[6] NPM package was used to load the .env file into the process environment variables

3.2.2 Database Setup

For designing the database, PlantUML was utilised—a tool for creating diagrams from text. It supports the creation of various diagrams, but for this use case, supporting ERD (Entity Relational Diagram) was sufficient. The final ERD describing the database layout can be seen on the next page in Figure 3.1.

For local development, docker-compose was selected as the widely accepted standard for running containerised applications, primarily, to spin up a PostgreSQL database container. To create the tables, indices, etc., an SQL `dump.sql` file was created, which contains the SQL initialisation commands. During the development, the `docker-compose.yml` file was augmented to mount the `init.sql` file into the database's `/docker-entrypoint-initdb.d/init.sql` file. By doing so, the SQL initialisation script was executed each time the database was created, ensuring that each time the volume was recreated, the required tables and data would be recreated as well.

6. https://www.npmjs.com/package/dotenv

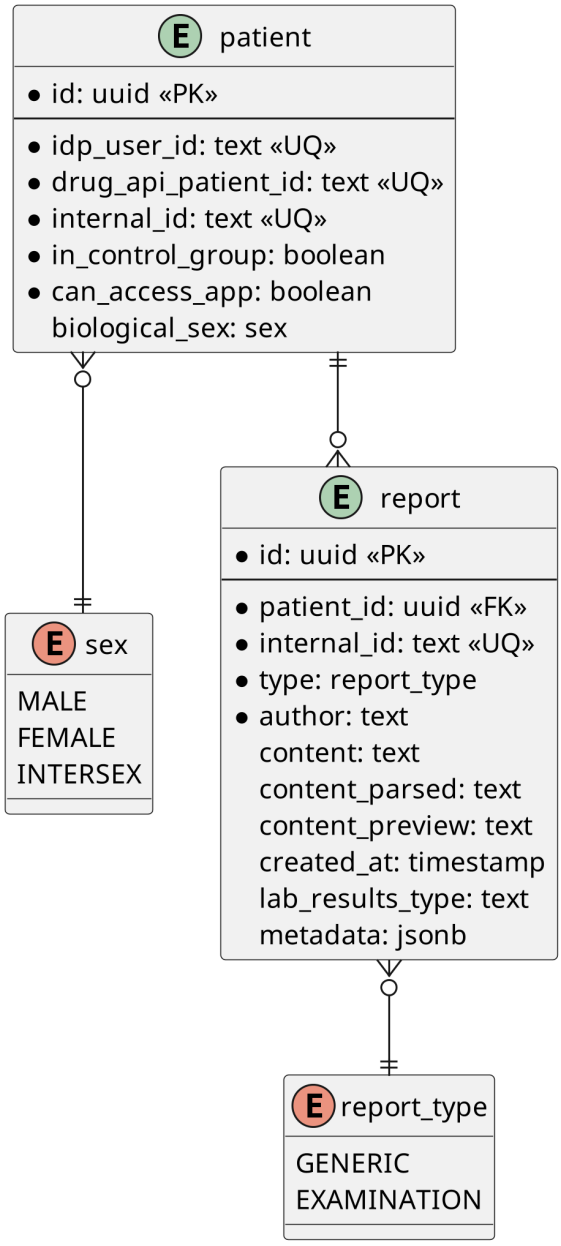

Figure 3.1: ERD Diagram

3.2.3 gRPC Client Integration

Since this service has to be able to communicate with the aforementioned Drug API service, a gRPC client had to be integrated. To achieve this, the `@grpc/grpc-js`[7] NPM package was utilised. A single class for handling all gRPC requests was implemented. During the initialisation of the client instance, it loads the `drug_api.proto` file and, successively, the package definition and descriptors. Finally, it connects to the Drug API service using the host provided through environment variables. Since each request requires an API key and a request ID in metadata, a function for creating a metadata object for each request was implemented. This function uses an API key provided in environment variables and generates a new UUIDv4 request ID. A unique request ID can be generated in this service, as each request sent to the Drug API service from here originates here as well.

3.2.4 Patient Registration

Before implementing any patient management API, an API for registering the patient had to be created. Several approaches to patient registration were considered before ultimately deciding on a particular method. The method chosen for use was primarily influenced by the fact that the patients should not be able to register themselves in the application but instead registered by a doctor during the visit. As a result, a POST method for creating the patient account was implemented. This method requires a few fields to be provided; the ones worth mentioning are the password and the patient's internal ID. The password is generated on the frontend during the registration, printed and handed over to the patient. The patients are encouraged to change their password as soon as possible. The internal ID is an ID used in the XML patient reports; this field is especially important for properly linking the data from the reports to the patient account. The whole process of registering the patient upon receiving the POST request in the Patient API is as follows:

7. https://www.npmjs.com/package/@grpc/grpc-js

1. As the request data has to be validated first, the NPM packages `class-validator`[8] and `class-transformer`[9] were utilised to ease the validation. Subsequently, these packages were also used later on in other methods for data validation.

2. Upon the validation, the patient has to be registered in the FusionAuth service. FusionAuth service also provides RESTful API, making it possible to create a user externally. In addition, it also offers a simple and lightweight HTTP TypeScript client, making it straightforward to use its API. However, an API key is required to authorise a restricted request to the FusionAuth service, which had to be created and provided to the Patient API using an environment variable. Finally, a request to register the user can be sent.

3. The next step involves registering the patient in the Drug API service. Since the Drug API not only handles drugs itself but also connects a patient with their assigned drugs, a Drug API patient account needs to be created. With the gRPC client already implemented, the process only had to call the specific gRPC method and provide the patient's FusionAuth ID.

4. After a successful patient registration in the FusionAuth service and the Drug API service, the patient details, along with some additional data about them, could finally be stored in the Patient API database. Executing this step after the two previous registrations was necessary since the ID from the FusionAuth and the Drug API service had to be stored alongside the patient data in the Patient API database. This action had to be done to establish a connection between the patient entities among multiple services.

5. Finally, the user, who was initially registered in the FusionAuth service without any direct link to the patient in the Patient API or the Drug API service, underwent an update in the FusionAuth service. This action, despite not being necessary, was done to include patient IDs from the Patient API and Drug API

8. https://www.npmjs.com/package/class-validator
9. https://www.npmjs.com/package/class-transformer

services in the FusionAuth, as it would simplify the implementation when connecting the patients between the three services.

3.2.5 Patient Reports Workflow

Once the patient registration flow is designed, the patient reports can be linked to the corresponding patients. As mentioned previously, the workflow to annotate and process the reports is complex, so an Apache Airflow DAG was defined. The DAG was created in Python, as recommended by the Apache Airflow documentation, and several tasks were defined.

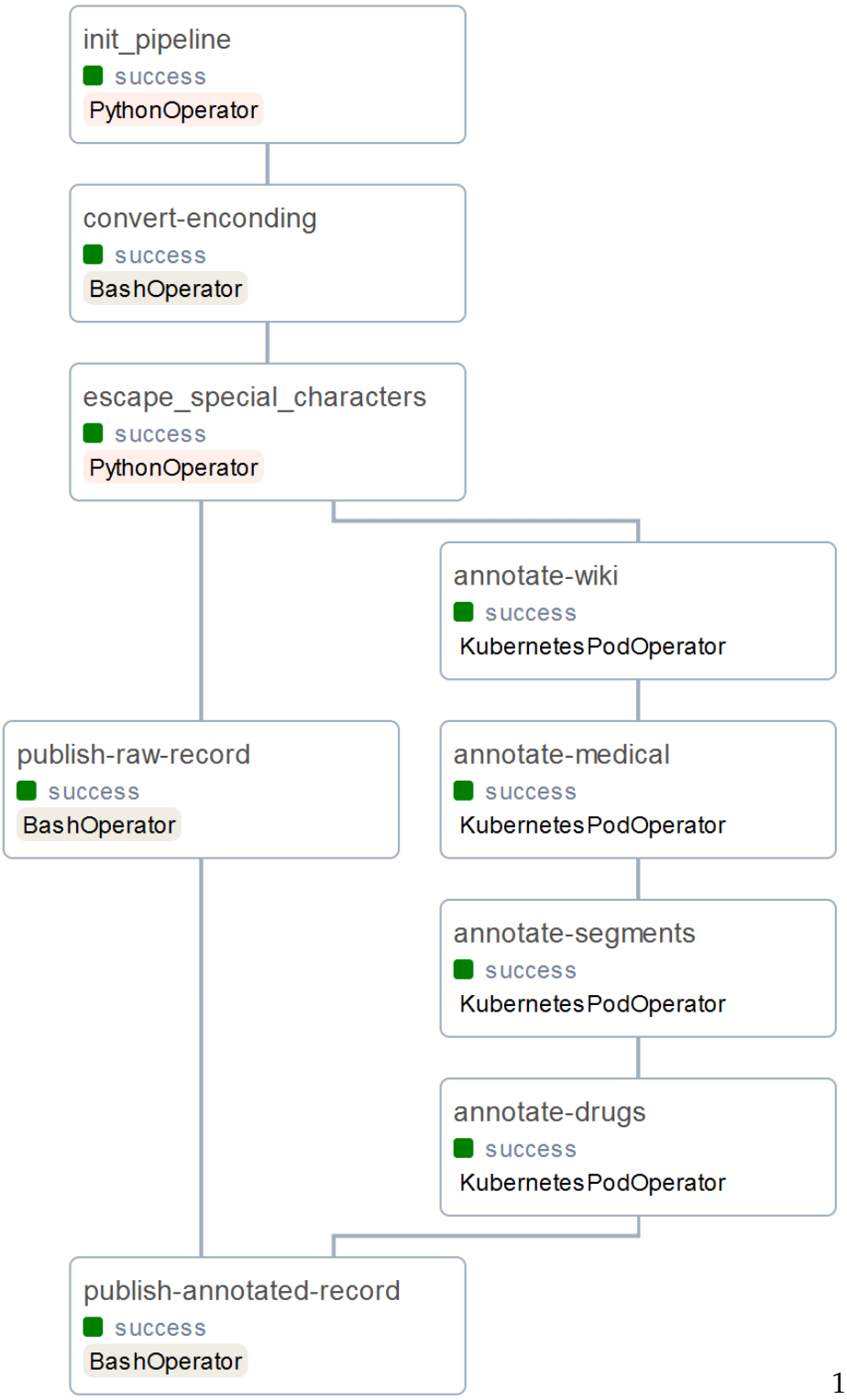

Figure 3.2: DAG visualisation

As shown in Figure 3.2, you can see the defined DAG with all its tasks. The DAG starts with the `init_pipeline` task, which, as the name suggests, initialises the pipeline. More specifically, it reads the file name from the context provided when the DAG was triggered, removes the file extension, and pushes the extension-less file name to XCom. XComs serve as a communication method that enables tasks to interact with one another. In our DAG, XComs are used to pass the extension-less file name between tasks, so each task knows which file it should operate on. Furthermore, each script stores the result in a temporary file that is later read by the following task; therefore, in case the directories for these temporary files do not exist, this task will create them. This task was implemented as a Python function with the PythonOperator in the same file as the DAG, due to its need to interact with Airflow DAG.

The next task converts the encoding of the original XML file from Windows-1250 to UTF-8, which ensures compatibility throughout the pipeline since UTF-8 encoding is a universal standard. Given the simple nature of this task, it was implemented as a bash script and defined using a BashOperator in the DAG.

The third task in the pipeline escapes all special characters in the report texts. The raw XML report files contain the report text wrapped in a `<![CDATA[]]>` section, which ensures that the text within this block is treated as regular text and ignores all special characters. However, since the annotators need to annotate the report text, the text must be parsed as regular XML and not as plain text. Removing the CDATA wrapper would render the XML file invalid since it already contains some special characters within the CDATA section. Therefore, I have implemented this task to escape all special characters found in the report text. This ensures that removing the CDATA wrapper does not invalidate the XML file. This task was implemented in Python using regexes since implementing a more extensive transformation would be unnecessarily complex in Bash, and thus it was defined using PythonOperator.

At this point, the flow splits in two and executes the following tasks parallel to each other. The right branch consists of four annotating tasks, each annotating something different. These tasks were created externally from this repository, and given their complexity, a docker image was built for each of them. Therefore, these tasks were

defined using the KubernetesPodOperator. This more complex operator runs the tasks by spinning up a container with the docker image that was specified. This container is created within a new pod in a Kubernetes cluster. Conversely, the left branch consists of a single task, `publish-raw-record`, which sends a POST request to Patient API and uploads the unannotated XML report file.

After both branches finish, the flow will join again and execute the last task, `publish-annotated-record`. Similarly to the previous publish task, `publish-annotated-record` sends a POST request to Patient API and uploads the annotated XML report file that was produced by the last annotating task. Both these publishing tasks are implemented in Bash and thus defined with BashOperator.

To summarise, to create a new patient in the system and store all the information we need, the patient must first be registered using the endpoint in Patient API to create a user in FusionAuth, Patient API, and Drug API databases. After the user is registered, a data pipeline can be triggered to process their XML report file and store any additional data available in the file.

As my own work goes, out of the whole pipeline, I have defined the DAG file and created the initialising, encoding, escaping, and publishing scripts. I have also contributed to implementing the annotators, specifically in the "interface" of the annotating scripts, mainly to provide a more uniform way of starting and configuring the annotators.

3.2.6 Report Processing in the Patient API

As mentioned previously, the pipeline includes two tasks for publishing both the unannotated and annotated report files to the Patient API. These two endpoints in the Patient API, where the files are being sent, serve one primary purpose—to finish some final processing and store the data in a database. The process is more complex than it may sound; let us start with the endpoint that handles the unannotated report files.

3.2.7 Unannotated Reports Handler

The method responsible for handling the request must first read the data from the file and then parse them so that it can interact with them

better. `@xmldom/xmldom`[10] NPM package was chosen for this job, as it offered everything needed—a simple and convenient DOM parser capable of parsing and serialising XML documents. Having the file parsed, basic patient data are then extracted, which is currently only an internal ID of the patient and their biological sex, but in the future, if necessary, significantly more data could be extracted. As mentioned earlier, the patient whose report file was just received and parsed needs to already exist in the system. So, now the process can patch the existing user in the Patient API database and provide only the new information gained from the file. Since the internal ID from the file is required during the registration, this process can use the internal ID to refer to the user in the Patient API database and provide the new data. The next step is to parse all the reports within the XML file. The file contains two types of reports:

- **Examination** — A report summarising the result of a specific examination.
- **Generic** — A generic report written when a patient visits a doctor.

These two types of reports differ in content, naturally, but also exhibit structure-wise differences. Considering the inconsistent shape, two individual extractors for both report types had to be implemented. The data being extracted are:

- **Type of the Report** — This is done to differentiate the reports in the future.
- **Internal Report ID** — Since it should be possible to re-run the pipeline on the same file in case the annotators change, it had to be possible to connect a potentially existing report in the database with a freshly parsed report from the file. Given this connection, if a report with such an ID already existed, it could be patched instead of creating a duplicate.
- **Report Creation Timestamp** — The date and time when the doctor wrote the report.

10. https://www.npmjs.com/package/@xmldom/xmldom

- **Author** — The name of the doctor who wrote the report.
- **Content Preview** — Since the full text with all the annotations that will be parsed and stored in the second endpoint is significantly larger, a preview of the report text was stored as well. This is useful when listing the reports since it does not have to fetch all the data but the previews, which are the first 220 characters.
- **Type of the Examination** — Another essential data is the type of the examination performed. Naturally, this information is lacking in the generic reports. Therefore, for these kinds of reports, the field is null.

After the reports are extracted, the process continues to store them in the Patient API database. Since the internal ID is available, it attempts to update the report with such an ID first and creates a new one in case it does not exist. Finally, it responds to the request with a list of created reports.

3.2.8 Annotated Reports Handler

This endpoint must be called after the previous endpoint for unannotated reports since it requires them to exist in the database already, as it only patches them by adding new data retrieved from the annotations. Similarly to the previous handler, it starts by reading and parsing the received annotated XML report file. Then, it proceeds to extract the annotated report data, which is achieved by using the same extractors as before. Nonetheless, it also extracts additional data that were not available in the unannotated reports, which are:

- **Internal ID** — Even though this was available in the unannotated reports, the handler still requires it to patch them correctly.
- **JSON-Parsed Annotated Content** — This represents the annotated XML text content parsed as JSON. For the frontend, it is beneficial to parse the XML content to JSON so it can traverse it and work with it more conveniently. Since the XML parser that was being used until now is a DOM parser and does not provide a convenient JSON object that can be easily serialised

and then de-serialised, a different parser had to be utilised for this job. NPM package `@rgrove/parse-xml`[11] appeared as the perfect tool considering the requirements.

- **Annotated Content** — Since the JSON-parsed content is complex and not so readable for humans to understand, pure XML annotated content was also included for debugging purposes. This field will be removed in the future.
- **Report Metadata** — One of the annotators also creates a new, more complex tag outside of the text, which collects metadata of the report and summarises them in the new XML tag. This tag also had to be parsed and converted to a more convenient JSON object.
- **Drugs** — The last drug annotator annotates all drugs within the report text; these drugs are now located and converted to a more appropriate structure for further processing.
- **Report Creation Timestamp** — This was also available in the unannotated reports; however, this time, it is required for drug registration in Drug API.

After being parsed, the reports can now be patched in the Patient API database. The internal ID is used to refer to the existing reports to patch them correctly. Afterwards, the next step is to assign the found drugs to the patient.

Assigning Patient Drugs

The next step is to assign the found drugs to the patient. The Drug API provides a `BatchCreate` method inside the `PatientDrugService`. My vision was to use `BatchCreate` and assign all drugs found among all the patient's reports. The `BatchCreate` input expects two fields, patient ID and a list of patient drug "items", which are objects consisting of drug ID and `created_by` string. Even though this appears correct at first glance, it lacks the essential `created_at` field. This is a field that the regular `Create` method does expect. The current Drug API

11. https://www.npmjs.com/package/@rgrove/parse-xml

implementation of `BatchCreate` is missing the `created_at` field, and instead, it uses the current time of when the request was received. It is not a critical bug, nor is it hard to fix; however, it is still present at the time of writing this text. To work around this issue, I have decided to send several requests using the regular `Create` method until the existing bug is fixed. Even though I have found a way around this problem, several aspects remain to be addressed.

First of all, the patient ID the method expects is not yet available to the handler. Nonetheless, this can be resolved by sending a simple request to the Patient API database to retrieve the primary patient ID using their internal ID.

Furthermore, the method also expects the `created_by` field to represent the entity that assigned the drug to the patient. In this case, it is the report in which the drug was mentioned. The format in which the method expects this field is `<resource-name:uuid>`, for instance, `report:f668ab95-8758-437b-b0d1-8e5667904e31`. Again, the issue lies in the fact that the handler only has access to the report's internal ID, not its primary ID. Fortunately, there is no need for further requests, as the preceding report patch provides a response containing the entire report, including its primary ID. Therefore, the internal IDs extracted from the file only had to be mapped to the primary IDs retrieved during the report patch.

Moreover, since the drug annotator annotates each drug it finds, in case the drug is mentioned multiple times in a single report, it would be extracted and registered numerous times. This poses a problem, considering it would lead to unnecessary duplication of drugs in the Drug API database, thereby causing confusion when the same drug appears multiple times in a single report.

Lastly, since the pipeline can be triggered again with the same report file, the drugs being registered should not be duplicated. Deleting all previously assigned drugs by the specific report appeared as the simplest solution. However, this has not yet been implemented, so it is subject to future development.

Therefore, the `BatchDelete` method was used to hard delete all drugs created by each of the extracted reports. Afterwards, the drugs could finally be registered using the `Create` method.

3.2.9 Rest of the API

The rest of the API only provides the patient and report data whose creation was covered in previous sections. The patient controller offers three routes for querying data:

- **GET** `/patient/:id` — Retrieves the patient identified by their ID in a URI parameter.
- **GET** `/patient/:id/report` — Fetches reports of a specified patient filtered based on query parameters, specifically on the `internal_id`, `type`, and `created_at`.
- **GET** `/patient` — Retrieves a list of all patients, with the ability to apply filters such as `biological_sex` and `idp_user_id` in the query parameters.

The report controller also offers three routes for querying data:

- **GET** `/report/:id` — Retrieves the report identified by its ID in a URI parameter.
- **GET** `/report/:id/patient` — Fetches the patient to whom the specified report belongs.
- **GET** `/report/` — Retrieves a list of all reports, filtered based on query parameters. These parameters are `internal_id`, `type`, `created_at`, and `patient_id`.

3.2.10 Documentation

To document the API, an `openapi.yaml` file was created. This file follows the OpenAPI Specification[12], which defines a format for describing REST APIs. It describes all routes, URI parameters, query parameters, request body, response body, error codes and possibly even more.

12. https://swagger.io/specification/

4 Gateway API

Gateway API is another pivotal service; it plays a crucial role in system architecture and design. This service's primary purpose/existence is to act as BFF (Backend for Frontend) and hide the implementation of other services consumed by the frontend application. Thanks to the Gateway API service, the frontend now benefits from a single service that offers a unified API, enabling seamless communication with other backend services. This kind of service can be replaced with a service mesh, like Istio[1] or Kong[2], or by incorporating an API management tool like Tyk[3]. However, in the case of this project, it was deemed infeasible. Primarily, due to numerous requirements on the gateway, such as implementing effective authentication and authorisation with the FusionAuth, integrating request transformers, incorporating gRPC service, etc. Implementing a custom gateway service enabled more granular control over data transmission and processing, facilitating the connection of data based on specific attributes and needs, while meeting all the previously stated requirements.

Primarily because our Gateway service had to be able to hide the implementation of the underlying gRPC service while at the same time implementing an effective authentication mechanism, which none of the previous tools could manage.

In summary, this service acts as an intermediary layer between the frontend and various backend services, dramatically simplifying the frontend's implementation. Without this service, the frontend would be required to directly interact with multiple backend services, each listening on various endpoints and utilising different communication protocols, while also imposing the responsibility of handling authentication on every single service behind the Gateway API.

1. https://istio.io/
2. https://konghq.com/
3. https://tyk.io/

4.1 Underlying Technologies

4.1.1 Language and Runtime

Since this service is just another web server application, without many language requirement differences from the previously implemented Patient API service, it was also implemented in TypeScript. The same applies to the runtime, and therefore, Node.js was chosen.

4.1.2 Framework

Given that the Gateway API service needs to communicate using more technologies than the Patient API service, and generally is more complex, a more versatile framework is preferred. Due to the previous experience, a natural choice was to use NestJS. It offers an easy-to-maintain, highly flexible, and scalable framework that seamlessly integrates with many libraries, tools, and services.

4.1.3 API

Due to the importance of efficient data handling and optimising resource utilisation, GraphQL API was a straightforward choice instead of the traditional RESTful API. One of the main features of GraphQL API is to let the client precisely specify what data are needed, which prevents over-fetching and under-fetching and alleviates unnecessary network traffic. The precision of selecting the exact shape of the data to fetch not only adds flexibility but also allows querying deeply nested structures in a single request, streamlining the frontend implementation.

GraphQL, as the name suggests, is a graph-based query language. Due to the API utilising a graph-based structure, the API-provided entities are interconnected. This interconnectedness allows for the querying of the nested structures as previously described. GraphQL API defines a set of entities and methods that can be executed upon them: queries, mutations and subscriptions. In this project, no subscriptions were utilised; therefore, the focus will be put solely on the queries and mutations. Queries are methods that only retrieve entities without altering any internal state of the application, while mutations are the exact opposite, as they are used for modification.

4.2 Implementation

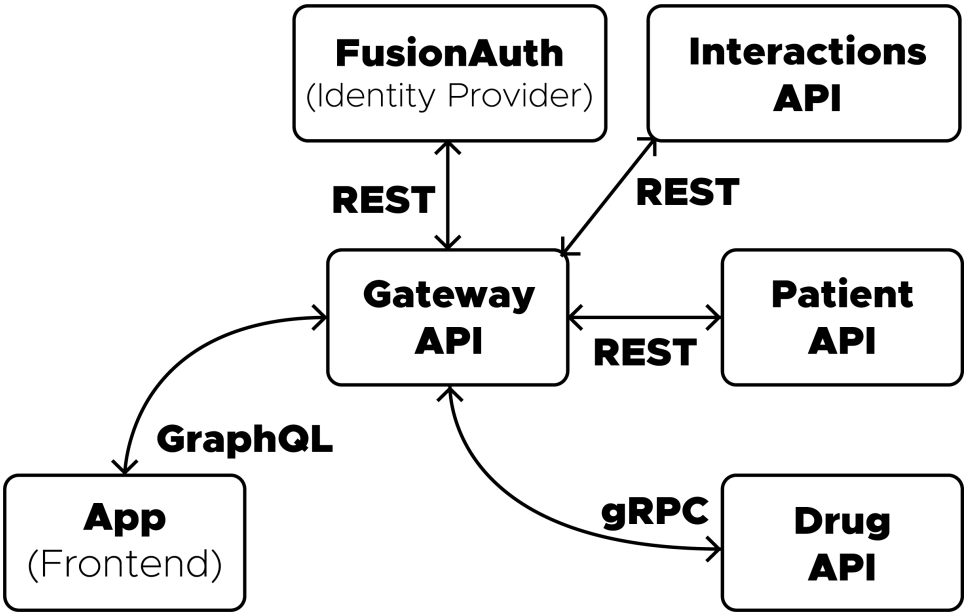

Figure 4.1: Gateway API communication diagram

Figure 4.1 shows all backend services hidden behind the Gateway API service. In this diagram, it can be seen how the frontend benefits from uniform API through which it can talk to each individual backend service.

In the following sections, the integration of each of the backend services will be explained, as well as the implementation of the core of the Gateway API service.

4.2.1 NestJS

To be able to delve deeper into the implementation of the Gateway API service, we must first understand how NestJS works.

According to NestJS documentation, NestJS, at its core, employs a robust Express HTTP server. Interestingly, it also offers the flexibility to incorporate Fastify, catering to diverse development needs. It stands out by providing an abstraction layer above these foundational frameworks while simultaneously offering access to their API for its

full potential. By utilising this approach, the vast array of third-party modules available for these frameworks can be used in NestJS as well [5].

Since NestJS provides an abstraction layer, several fundamental blocks used to build the NestJS application need to be explained. The most essential building block is a provider, which can be any class that offers some service or logic. The top-most fundamental block in the abstraction is a module. Modules provide metadata regarding sections of the application, guiding NestJS on how to structure and organise the application architecture. This metadata may contain various details, such as a list of providers included in the current module, imported modules that need to be accessible within it, and providers that are exported from the current module, making them available for use in other modules. A module can be thought of as a collection of providers that are tightly logically related to each other and together create a bounded context known from a DDD (short for Domain-Driven Design) context[6].

NestJS mainly relies on dependency injection. Dependency injection is a technique that strives to create loosely coupled applications by extracting the process of internal creation of dependencies, and in the case of NestJS, by delegating this concern to the framework itself. Thanks to the metadata provided in modules, these dependencies can now be dynamically injected into places where they are needed.

Creating base application

To start with the implementation, the base of the application had to be created. NestJS provides a handy CLI tool[4] that assists with the initialisation of a project, the boilerplate generation when creating new modules, providers, etc., and finally, building and bundling the project for production deployment. Running this initialisation command generates a skeleton of the application with several files, the most important ones being `main.ts` and `app.module.ts`. The `main.ts` file represents an entry point for the application, where the NestJS application is created using a `NestFactory`. The second most important file is the app module, the root module that wraps the whole

4. https://www.npmjs.com/package/@nestjs/cli

application. The rest of the created source files exist solely as an example implementation of a RESTful API implemented in NestJS, and therefore, they were removed.

4.2.2 Configuration

To configure this service, just like with the Patient API service, a `.env` file was employed, as it is a common strategy in Node.js applications. Another benefit of NestJS is a built-in ConfigModule, which automatically reads `.env` files and provides a service to access this configuration. To make use of this module, it was registered in the root app module as a global module. Registering it globally results in the module being available in all other modules without the need to import it explicitly.

4.2.3 GraphQL

NestJS, due to its versatility, already offers a module for GraphQL. It is built on top of the Apollo GraphQL platform[5], with the option to use the second most popular GraphQL platform, Mercurius[6]. The Gateway API service adhered to the default setting and, as a result, is using the Apollo platform.

NestJS offers two approaches for implementing GraphQL applications: code-first and schema-first. The schema-first approach revolves around defining the GraphQL/SDL schema first, which is used to define the API. It is a fundamental element in GraphQL, as every GraphQL client typically consumes it to generate classes and types from this file, simplifying interactions with the API. However, defining the SDL schema first means that all resolvers must match exactly the previously defined schema. As such, this results in a harder-to-maintain codebase and generally is less scalable. On the other hand, the code-first approach involves defining TypeScript classes and using decorators to add metadata to each field, method or class. NestJS then uses these decorated metadata to generate the resulting SDL schema. One of the limitations of the code-first approach is the risk of focusing excessively on the execution rather than the actual requirements

5. https://www.apollographql.com/
6. https://mercurius.dev/#/

of the API. Conversely, prioritising the API's design from the beginning, as seen in the schema-first approach, often results in creating a more thoughtfully structured API [7]. For our project needs, the code-first approach was deemed preferable, primarily due to its better maintainability.

To incorporate this GraphQL module in the application, it had to be imported into the root app module and configured afterwards. The aforementioned config service was employed to configure this module. The GraphQL CORS setting was enabled, and the allowed origin was configured based on the value provided by `ConfigService`. Another important setting is the environment in which the application is being deployed: development or production. In the development environment, the GraphQL introspection and a playground were enabled, while in the production environment, none of these features were enabled. These features were not enabled in the production due to the possibility of posing a security risk. GraphQL introspection allows the client to ask for the structure of the API, the queries and mutations it provides. Playground is another useful feature used during the development; it provides a useful UI for testing the API. Later on, this default playground was replaced by an embedded Apollo server, which offers functionality similar to the playground but extends it significantly. This replacement was carried out primarily due to the need to measure and trace the times of the individual requests processed by the API.

NestJS abstracts the GraphQL request handling into two providers: resolvers and services. GraphQL resolvers define the available queries and mutations, their inputs and outputs, descriptions and other metadata used for configuring these methods, while GraphQL services provide the functionality, like retrieving and manipulating the data.

4.2.4 API Graph

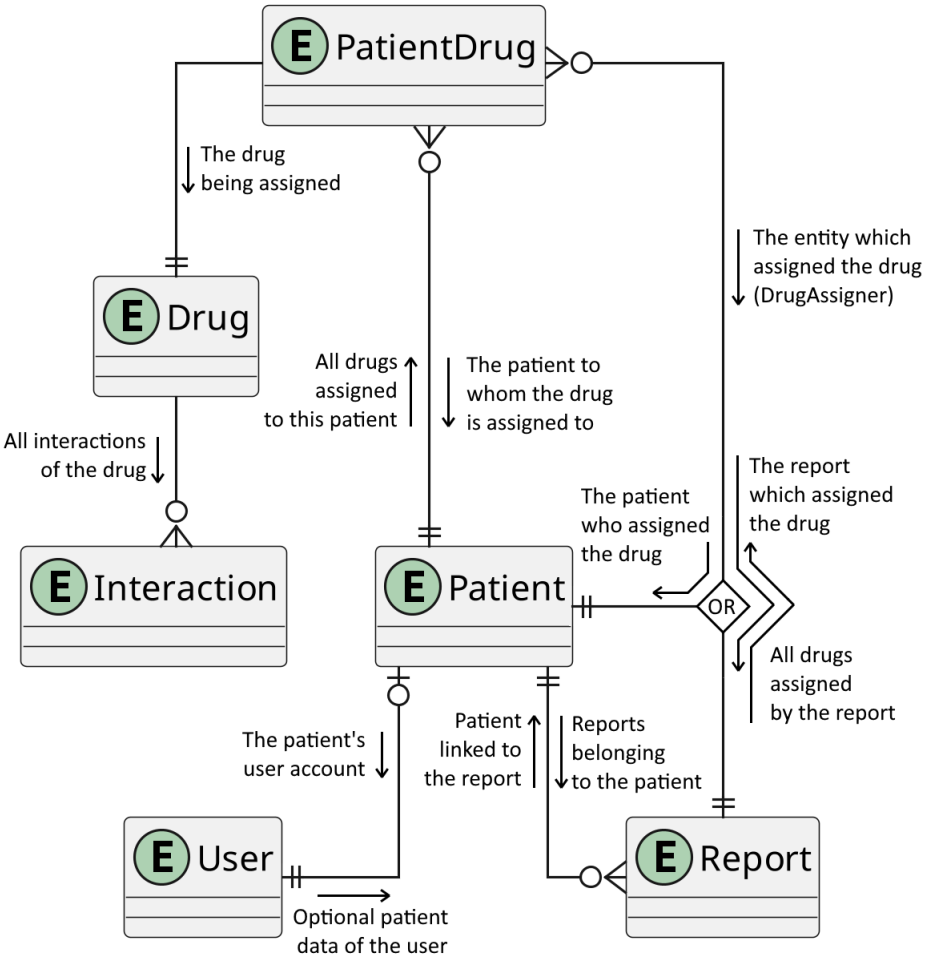

Figure 4.2: API graph structure

Figure 4.2 above describes the resulting graph the API provides. The actual data stored within the entities are omitted for simplicity purposes, and only the relations are displayed. Single and semi-directional relations are also displayed; for instance, `PatientDrug` has a reference to `Drug`, but not vice versa. Again, this graph clearly portrays the benefits GraphQL and the Gateway API bring. Frontend is able to send a single request to the Gateway API in which it asks for data about

a user (retrieved from FusionAuth), their patient and report data (retrieved from Patient API), their assigned drugs (retrieved from Drug API), and finally interactions between these drugs (retrieved from Interactions API).

4.2.5 Drug API Integration

Drug API was the first service integrated into the Gateway API service. The NestJS CLI tool was used to generate the boilerplate code when creating a new resource module for GraphQL. Resource modules provide functionality for working with specific resources, in this case, drugs. The generated `DrugsModule` includes `DrugsResolver` and `DrugsService`.

As previously mentioned, the Drug API service provides its API using the gRPC technology, so a gRPC client had to be incorporated. Fortunately, NestJS provides a module for gRPC microservice. This module was imported into the `DrugsModule` to make it available for its providers, namely `DrugsService`.

Since the gRPC API is defined in a protobuf file, TypeScript interfaces describing this API had to be created. This process involved analysing the services and methods defined in the protobuf file and creating their equivalent in TypeScript. This process had to be repeated each time the API changed, and therefore it was deemed unsuitable. As a workaround, the NPM package `nestjs-proto-gen-ts`[7] was employed. This package can generate the TypeScript interfaces automatically based on the protobuf file. However, even this solution was not perfect; each time the API was changed, the protobuf file had to be manually downloaded, and the generator executed. Another attempt to resolve this issue was automating this process by incorporating the generation of the interfaces in a CI/CD pipeline while building the Drug API service. The resulting interfaces were then packaged as a standalone NPM package and published. This solution streamlined the update process tremendously, at least in theory, as the only remaining task was to update the package's version number in the dependencies list. During the development, another problem arose. However, this time, the problem was not related to automation or

7. https://www.npmjs.com/package/nestjs-proto-gen-ts

repetitive manual work; instead, the generated interfaces were unusable. The protobuf version used in the Drug API service did not mark optional fields as optional because it was impossible in older protobuf versions. Beginning from the version v3.15.0 and onwards, the optional keyword became available. However, due to the lack of the optional keyword, the package generating the TypeScript interfaces could not detect optional fields. As a result, it treated each field as mandatory, even though it was optional. Consequently, methods containing optional fields were not usable since the generated interface required all the fields to be provided. Temporarily, the solution was to keep using the NPM package at least to provide the protobuf file and avoid having to download and replace it manually. The interfaces, for now, are maintained separately; each time the API is changed, the interfaces are updated according to the generated interfaces, with the exception of manually marking optional fields as such. In the future, a better strategy could be employed to solve this problem, potentially updating the Drug API service to mark optional fields properly, allowing the generator to recognise these fields and generate the resulting TypeScript interfaces adequately.

The `DrugsResolver` was implemented to provide two queries: `drug` and `drugs`. While the `drug` query expects a drug ID and returns the specified drug, the `drugs` query expects a filter object that contains fields to narrow down the result. These queries use the gRPC client and retrieve the desired data using the `DrugService.Get` and the `DrugService.List` methods provided by the Drug API.

Configuration

Similarly to how the gRPC client was configured in the Patient API service, a host and the API key to connect to the Drug API had to be provided. These values were retrieved from the config service.

4.2.6 Patient API Integration

The second service integrated into the Gateway API was the Patient API. Due to the fact that the Patient API provides two resources, namely a patient and a report, three separate modules were created. The first module `PatientApiModule` provides the functionality of com-

municating with the Patient API service. Unlike the previously integrated Drug API service, the Patient API service offers RESTful API. As a result, it internally utilises an `HTTPModule`, provided by NestJS, to allow communication with the Patient API service. The other two modules can be seen as GraphQL resource modules, namely `PatientsModule` and `ReportsModule`, containing the resolvers and services.

In summary, the `PatientApiModule` is responsible for the actual data retrieval from the Patient API, while the other two modules are in charge of providing this data to the client.

The `ReportsResolver` defines two queries: `report` and `reports`. Similarly, the `PatientsResolver` defines two queries `patient` and `patients`, along with a mutation `createPatient`. This mutation employs the aforementioned POST method in the Patient API for registering new patients.

However, since the resolvers only provide the base object by default, supplementing field resolvers was necessary to allow the client to request nested structures of related entities. A field resolver is a function within a resolver that provides the functionality to fetch the related entity. As an example, if a client asks for a patient with a specific ID, they can also ask for all the reports belonging to this patient. The `PatientsResolver` first resolves the patient itself, then employs the field resolver to also resolve all their reports.

To incorporate this feature, two field resolvers were introduced, one for resolving the patient associated with a report and one for resolving the reports belonging to a patient.

4.2.7 Patient API and Drug API Interaction

As previously mentioned, the Drug API also assigns drugs to a patient. A field resolver can be implemented in the patient entity to reflect this interaction in the Gateway API. However, due to the fact that the drug assignment also contains some important metadata, like the time the drug was assigned and the entity that assigned the drug, a new resource has to be created: the PatientDrug entity. A PatientDrugsResolver and PatientDrugsService were created within the DrugsModule to structure the application properly. The `PatientDrugsResolver` defines three field resolvers:

- **Patient field resolver** — Resolves the patient to whom the drug is assigned.
- **Drug field resolver** — Resolves the drug that is being assigned.
- **DrugAssigner field resolver** — Resolves the entity that assigned this drug.

Currently, the entity responsible for assigning drugs can be either the patient themselves or the report in which the drug was mentioned. Plans are in place to enable doctors to assign drugs in the future. Furthermore, given that the drug assigner entity can take the form of a patient or a report, a GraphQL union type was created. In GraphQL, union types are specifically used for this purpose—to group possible types into a single one. This relation is more clearly illustrated in the visualised graph in Figure 4.2.

The `PatientDrugsResolver` not only offers a query, `patientDrugs`, for requesting a list of patient drugs filtered according to an input, but it also includes two mutations: `reportDrugUsage` and `stopDrugUsage`. These mutations enable patients to either assign a drug to themselves or remove it.

4.2.8 Interactions API Integration

Integrating Interactions API was straightforward as it offers just one API method. This method requires an ATC code of a specific drug and returns a list of ATC codes for drugs that interact with it, detailing the potential effects of these interactions. Additionally, the API provides a list of food interactions, including the effects they might cause.

Initially, the plan was to send a request to the Interactions API to resolve all interactions of a specific drug, resulting in a list of ATC codes. These ATC codes were then planned to be further resolved to particular drugs the client could ask for. However, due to multiple drugs sharing the same ATC code, this approach was not suitable. As a result, the returned interactions object only lists the ATC codes of the interacting drugs along with the effects they might cause.

To integrate this service, an `InteractionsModule` was created along with its resolver and service. The resolver contains only one query `interactions`, which expects an ATC code and internally resolves

the interactions of the given drug. Moreover, a field resolver was also incorporated into the `DrugsResolver` to provide the interactions of the specific drug.

4.2.9 Authentication

Estabilishing authentication as a fundamental component of the Gateway API service ensures that authentication is resolved before reaching the other backend services. This approach relieves these services from handling authentication, allowing them to concentrate solely on delivering their intended functionalities.

FusionAuth's hosted backend handles the process of logging in to the system. After a user successfully authenticates in FusionAuth, a JWT token is issued and stored in a cookie. This JWT token is then sent with each request to the Gateway API, and it is now up to the Gateway API to verify the validity of this token.

JWTs (short for a JSON Web Token) are encoded and signed tokens containing claims about a user. Given that anyone can create and encode a token, potentially impersonating someone else, it is necessary to verify these tokens to prevent tampering. Each token contains a signature at the end, generated using a secret key. Without access to this key, forging the signature is not possible.

Integrating authentication in a regular Express application would involve creating a middleware, while in NestJS, guards are available. Guards are specialised middleware for detecting whether a request can continue to the method handler. Compared to the middleware, they are more advanced, as they have access to the execution context and can act accordingly. These guards can then be applied to individual queries and mutations, automatically triggering the authentication process and not allowing unauthenticated users to go any further.

The Gateway API's authentication guard implementation involved extracting the JWT token from a cookie, verifying its validity, and storing the decoded JWT payload representing the authenticated user inside the request object. NPM package `cookie-parser`[8] was employed to parse the cookies, while `JWTModule`, provided by NestJS, was used to validate the token. This module provides functionality for verify-

8. https://www.npmjs.com/package/cookie-parser

ing the JWT token's validity. Since many algorithms may be used for signing and verifying JWT tokens, the `JWTModule` had to be configured accordingly. FusionAuth was configured to issue JWT tokens signed with an RS256 algorithm. To properly configure the `JWTModule`, the RS256 public key, retrieved from the FusionAuth and provided through the config service, was used, along with the JWT issuer, which is "aicope".

4.2.10 Authorisation

Authentication is not the only security concern handled by the Gateway API, as it also performs authorisation. The FusionAuth service defines user roles, and each user can have a subset of all defined roles: admin, doctor, patient and survey respondent. Currently, these roles are treated like a level of privilege, rather than roles. This means that users with the doctor role can access methods guarded with the patient role since the doctor role has a higher privilege. This behaviour is planned to change in the future to convey the proper meaning of roles.

To properly authorise users, a `UserRoleGuard` was created. It depends on the success of the previously implemented `AuthGuard`, and as such, before continuing, it executes this guard and continues only after a successful authentication. The user's roles are extracted from the decoded JWT payload stored in the request object, and only the highest role is considered. The guard was parametrised to authorise any user with at least the specified role required for the specific query or mutation. If no role was specified for the query or mutation, the guard allowed the execution to continue to the handler if the authentication guard succeeded. Finally, the resulting `UserRoleGuard` was globally applied to all queries and mutations without any role requirement, effectively rendering the whole API available only to authenticated users. Queries and mutations requiring a specific role were explicitly marked to use the role guard with the necessary role. The majority of the API required the user to have at least the patient role; however, several queries, like the ones providing information about drugs, are available to any authenticated user.

4.2.11 Users

With the authentication being finished, the last module for users can be implemented. `UsersResolver` offers a single, although very important, query. This query, carrying the name `me`, provides information about the currently authenticated user. With this query, patients get access to nearly the whole data graph described in Figure 4.2. However, this would not be possible without the essential field resolver for resolving the patient linked to the user. This link is available only if the current user is a patient.

5 Deployment

It can be quite challenging and complicated to deploy a project of this size with many services. It was crucial to utilise some form of provisioning tool to ease the deployment, configuration, storage allocation, etc. Terraform, a leading provisioning tool[8], offers a declarative language for writing infrastructure-as-code; meaning the code defines how the resulting infrastructure should look, rather than defining the steps to reach the desired result. This tool was used to provision and manage the FusionAuth service, the database, and many Kubernetes secrets for managing sensitive keys in a secure manner. However, the most notable service benefiting from the use of this tool is FusionAuth, considering it required a significant amount of configuration that varied from the defaults.

I have personally contributed to provisioning and managing the infrastructure, defining the desired state of several Terraform resources, specifically those configuring features in the FusionAuth that were required by the Gateway API or the Patient API services. For instance, to grant the Gateway API and the Patient API services access to the FusionAuth, API keys were created and automatically populated into Kubernetes secrets, for these services to load.

I have also created docker build files, for building images that can be easily deployed in the existing infrastructure. Additionally, helm charts were also created by others, as a configuration for the packaging eco-system for Kubernetes [1], complementing my own work with the build files.

The system operates within a secure cloud environment tailored for scientific research. It is built on top of Kubernetes infrastructure[2] equipped with GPU-enabled hardware, specifically for running ML (short for machine learning) tasks and accessing highly secure or compliant data [9]. Additional details are further elaborated in the previously mentioned David Rusnák's thesis [4].

1. https://helm.sh/
2. https://kubernetes.io/

6 Conclusion

This project strived to establish a solid foundation for an advanced patient empowerment application through a process of iterative system development. The invaluable feedback from doctors and patients at the MMCI played a critical role in refining the application into a state capable of deployment.

The primary aim of this thesis involved the development and integration of two crucial services: the Patient API and the Gateway API, into the existing AICope project. This integration was pivotal for fostering further advancements. Moreover, incorporating a third-party identity provider added an additional security layer, ensuring more secure management of patient data.

The successful achievement of the thesis objectives is evident as the application is now undergoing live testing in a real-world clinical environment. Reaching this stage of development is a significant accomplishment in itself. Yet, it's important to recognise that this is just the first step towards the ultimate goal. This journey has set the groundwork for future enhancements, signalling a promising direction for advanced patient care solutions.

6.1 Future Work

Throughout the thesis, various details were highlighted that require additional care to be taken. Still, these should be seen as just that—minor yet important aspects to consider. However, from a slightly different perspective, the Gateway API in its current state can be relatively inefficient in terms of data handling performance. This is due to the way GraphQL and its resolvers work by default. For instance, we can consider the following example. A patient requests a list of drugs that were assigned to them. The Gateway API has to send a request to the Drug API, asking for these drugs. However, the Drug API only outputs a list of patient drugs, which do no include any drug information, rather only their ID and several metadata of the assignment. Here is where the problem lies, the Gateway API, upon receiving the list of IDs, attempts to resolve them to the actual drug data, sending N requests to the Drug API to retrieve these drugs by

their IDs. Although only a single request for resolving all drugs is sufficient, that is not how the Gateway API handles it. It is a common problem in GraphQL, often referred to as the "N + 1 problem", as it sends N additional requests to resolve the N elements returned from the initial "+ 1" query.

A common solution for this problem is the use of data loaders. Instead of sending a request each time a specific entity is needed, data loaders accumulate these requests and send a single batch request providing a list of IDs for the accumulated entities. Data loaders often provide a caching solution as well, further optimising data handling performance in GraphQL [10].

Another topic that was not mentioned throughout this thesis is database migrations. They are an essential aspect in the development of modern applications, to ensure changes are controlled, tracked and reversible. As of now, the Patient API does not utilise database migrations, however, it is subject to future development. Afterwards, the drugs could finally be registered using the `Create` method.

Nevertheless, those are only minor planned adjustments to the backend services, which are unlikely to have any significant effect. More importantly, the long-awaited feedback from the patients after the live testing will be crucial. These results will provide indispensable insights, that will determine the future direction of this project.

Bibliography

1. PATERICK, Timothy E.; PATEL, Nachiket; TAJIK, A. Jamil; CHANDRASEKARAN, Krishnaswamy. Improving health outcomes through patient education and partnerships with patients. *Proceedings (Baylor University Medical Center)*. 2017, vol. 30, no. 1, pp. 112–113. Available from DOI: 10.1080/08998280.2017.11929552.
2. *Most Popular Technologies: Language - Stack Overflow Developer Survey 2023* [online]. Stack Overflow, 2023 [visited on 2023-12-11]. Available from: https://survey.stackoverflow.co/2023/#most-popular-technologies-language.
3. NIRNEJAK, Jitendra. *Express.js vs Fastify - In-Depth Comparison of the Frameworks* [online]. 2023. [visited on 2023-12-11]. Available from: https://www.inkoop.io/blog/express-vs-fastify-in-depth-comparison-of-node-js-frameworks/.
4. RUSNÁK, David. *DevOps and MLOps Pipelines in a Secure Kubernetes Cluster* [*online*]. Brno, 2023. Available also from: https://is.muni.cz/th/ori9x/. Master's Thesis. Masaryk University, Faculty of Informatics. Supervised by Vít NOVÁČEK.
5. NESTJS. *NestJS - Documentation* [online]. 2023. [visited on 2023-12-15]. Available from: https://docs.nestjs.com/.
6. FOWLER, Martin. *Bounded Context* [online]. 2014. [visited on 2023-12-18]. Available from: https://martinfowler.com/bliki/BoundedContext.html.
7. LO, Victoria. *GraphQL for Beginners: Schema-first vs Code-first* [online]. 2023. [visited on 2023-12-15]. Available from: https://lovictoria.com/graphql-for-beginners-schema-first-vs-code-first.
8. IBM. *What is Terraform?* [online]. [visited on 2023-12-18]. Available from: https://www.ibm.com/topics/terraform.
9. MASARYK UNIVERSITY. *Sensitive Cloud | Infrastructure and Services* [online]. [visited on 2023-12-18]. Available from: https://www.cerit-sc.cz/infrastructure-services/sensitivecloud.

10. APOLLO GRAPHQL TEAM. *How Federation handles the N+1 query problem* [online]. 2023. [visited on 2023-12-18]. Available from: `https://www.apollographql.com/docs/technotes/TN0019-federation-n-plus-1/`.

A Relevant Links

All relevant repositories for the implemented services, whether by me or someone else in the team, are available in the University's GitLab `https://gitlab.fi.muni.cz/bimedi/aicope/`.