A new approach to a digital healthcare management system

Remi Bahar
210947548
School of Electronic Engineering and Computer Science

July 2022

Abstract - after research, interoperability and GDPR-compliance were identified as key barriers to healthcare digitisation. The latter of which is unaddressed by existing solutions. This project seeks to expand on existing solutions by developing a proof-of-concept healthcare management system that addresses these barriers. In partnership with an industrial partner an MVP OData Patient service implementing interoperability was developed, followed by a healthcare management system implementing GDPR-compliance. Extensive testing was conducted to ensure the system met its requirements. Best practices were followed, including using a microservices approach.

Keywords - gdpr, oData, healthcare, digitisation, patient, api

1 Introduction

One of the main challenges facing healthcare systems is ageing population. The average life at birth is over 80 years in OECD countries. Over 80% of people ages 65 years and older are affected by chronic diseases. This has resulted in an increased demand for healthcare systems as chronic diseases such as diabetes are often long-term and require multiple interactions with health systems. Indeed, in the EU around 70-80% of a country's total health expenditure is on chronic diseases. Chronic diseases are also responsible for 86% of all deaths [4].

One way of addressing these issues is through healthcare digitalisation. Healthcare digitalisation is a broad topic, encompassing many areas such as mobile apps, big data, Artificial Intelligence (AI), and the Internet of Things (IoT). At its core, healthcare digitalisation can improve the quality of care by ensuring caregivers spend less time on administrative tasks and more time on patient care. Access to more information e.g. from wearable devices, or insights from AI can allow for more informed decision making. It can also reduce the strain on healthcare systems, by allowing virtual consultations between doctors and patients. This remote monitoring, is especially useful for patients in remote regions. Indeed, nearly 80% of surveyed doctors believe telemedicine is a better way of managing chronic diseases [4].

The aim of this project is to help drive healthcare digitalisation by developing a proof-of-concept healthcare management system that satisfies two key challenges faced by existing systems.

The first key aim is to develop a healthcare management system that is interoperable. Increased interoperability has the potential to increase the quality of care, with 69% of patients and 85% of healthcare professionals believing that health system integration can improve the quality of care for patients. Interoperability is a well-developed area with implementations such as HAPI-FHIR already having level 3 interoperability [8]. Despite this, interoperability issues are still prevalent with 74% of patients reporting that they have to repeat the same information to multiple healthcare professionals [4]. To help address interoperability, this system implements interoperability to a comparable level as existing systems.

The second key aim is to develop a healthcare management system that is GDPR-compliant. One of the top barriers to healthcare digitalisation is privacy and data security concerns, with 29% of healthcare professionals reporting that it is a top barrier [12]. These concerns are not unfounded. In 2020, Capio St. Göran's hospital received a 30 million SEK fine for not complying with GDPR regulations. Among other offences, the aforementioned hospital failed to limit employee access to medical records based on what was required in their job role [3]. Furthermore GDPR applies to the healthcare management system since it involves the processing of personal data in the form of personal medical records [16]. There is arguably a gap in existing systems with regards to GDPR-compliance. HAPI-FHIR does not define a security service [2]. While the NHS, does implement Role Based Access Control, activities are not linked to the data model making it harder to manage and audit the security system [6].

2 Related Work

Literature review was conducted by critically analysing existing systems in the context of two research questions based on the two key-aims of this project.

A How to ensure the healthcare management system is compliant with GDPR regulations?

One possible method of ensuring GDPR-compliance is that used by the NHS Personal Demographics Service FHIR API. This API uses the NHS national role-based access control (RBAC) table to authorise a user's request [5]. In this scheme users are assigned pre-defined roles which are linked to pre-defined activities on a many-to-many basis. This ensures GDPR-compliance specifically by adhering to data minimisation, as users should only have access to data relevant for their role. This can be illustrated in an example. Suppose we have a role, Patient administrator, linked to an activity, update Patient data. Users with a job title of Doctor might be assigned this role which will allow them to be authorised to update Patient data. This complies with GDPR regulations as part of a Doctor's role may be to update patient records e.g. to include notes about a consultation with the patient [6].

However one major shortcoming of this implementation is the difficulty linking activities to the data model. In the example given above, how do we define what fields the "update patient data" activity should correspond to.

The proposed implementation in this project aims to address some of these issues faced by the NHS RBAC. In the proposed system employees and/or job titles are assigned to roles on a many to many basis. Roles are then assigned on a many to many basis to an entity e.g. a Patient or even an individual field e.g. the Address of a Patient. This allows granular access control as the specific fields a role has access to can be controlled. The action element of Attribute Based Access Control (ABAC) can also be implemented by including am access-level with a role's mapping to a field. This access-level can be used to control what actions can be authorised e.g. by setting it to give read access or read and write access. This information can be persisted in a security database. An IAM OData service can be implemented for interacting with the security database [10].

The advantage of the proposed implementation is that individual requests can be inspected and compared with the security database. Suppose a user, e.g. an employee in HR, has access to the name, age, and gender of Patients but not their address. A request can be made to the IAM service with the user's ID to return a list of the fields they have access to. If using OData, this list of fields can be appended to their request Patients?$select=Name,Age,Address which is then forwarded to the Patient data service. This ensures the user only has access to specific fields they are authorised for. The disadvantage of this implementation is that it does not completely implement ABAC as it does not take into account the environment details of the request [10].

The baseline to compare the proposed system against for GDPR-compliance is the NHS Personal Demographics Service FHIR API.

B How to ensure the healthcare management system is interoperable?

Interoperability refers to the ability of different systems to exchange data in a coordinated manner. HIMMS defines 4 levels of interoperability.

- Foundational (Level 1): One system can securely exchange data with another system
- Structural (Level 2): Structure of data exchange is defined
- Semantic (Level 3): Standardises data and creates common terminology [8]

To answer this research question, two related systems, namely Fast Healthcare Interoperability Resource (FHIR) and the Open Data Protocol (OData) will be critically analysed in the context of this research question.

FHIR, developed by HL7, is a RESTful standard describing the format of data exchanged between medical systems. It achieves levels 1 to 3 interoperability by being RESTful, providing metadata, and predefining resources used in a healthcare system such as a Patient and their fields [7]. One implementation is HAPI FHIR in Java. In addition to implementing FHIR, it includes modules such as a JPA Server that can be used for the persistence and CRUD of personal medical records. This achieves the first two project aims as well as including some more advanced features such as database partitioning and batch processing. However the main shortcoming is that it does not define a security service needed to ensure GDPR-compliance which is a key aim in this project [2].

The Open Data Protocol (OData) defines best practices for RESTful APIs. Since it is based on REST it supports level 1 foundational interoperability. Furthermore, it represents the service's data model in a metadata document which can be viewed by the client using the system query $metadata. This provides level 2 structural interoperability.

It's main disadvantage compared with FHIR is it does not have as comprehensive semantic interoperability since it does not predefine resources such as a Patient which have standardised fields and data types in FHIR. However it defines powerful system queries allowing for more control over the data returned to the client by the server. Consider a scenario where a person is requesting patient data but only has authorisation to view the names of patients. Using $select an authorisation service can intercept the client's request and forward a request to the data service with Patients?$select=Name to only return the names of patients. This allows field level security to be implemented.

Another scenario may be where row-level access control is required. If an employee is working at a hospital, e.g. Ealing hospital, they may only have access to view patients at that hospital. Using $filter this can be implemented by Patients?$filter=Location eq Eailing Hospital which will return only patients at Eailing hospital. [9]

On-balance, OData was chosen for this project as it provides powerful system queries to help implement row-based and field-based access control.

The baseline to assess the proposed system against for interoperability is HAPI-FHIR, however only HIMMS foundational and structural interoperability are targeted due to time constraints with this project.

3 MVP Methodology

A Requirements

Requirements 1 and 5 were determined after conducting literature review to analyse the challenges facing healthcare digitalisation and critically analysing existing solutions. Requirements 2b to 4 were obtained from interactions with the industrial partner, cmd Consulting GmbH whom the prototype healthcare management system is being developed for. Initial requirements were obtained from the project brief [15]. These were expanded on by writing a survey and sending it to the industrial supervisor Franz-Josef Ziegert [17].

There are to be two releases of the proposed system. The first, referred to as the MVP (Minimum Viable Product), will meet the requirements set out in the project brief [15], the interoperability requirements detailed in section B, and the non-gdpr related requirements from the industrial partner survey [17]. This release is due by 15th July 2022.

The second release, referred to as the Extension, will meet the requirements set out in section A and by the industrial partner survey [17]. This release is due by 15th August 2022.

1. The system should be designed to be interoperable
   (a) Foundational (level 1) interoperability should be fulfilled [8]
   (b) Structural (level 2) interoperability should be fulfilled [8]
   (c) The OData protocol should be used with support for $metadata, $filter, $select system queries
   (d) The API should be RESTful, i.e. use GET for read, POST for insert, PATCH or PUT for update, and DELETE for delete
2. The system must meet the data model requirements set-out by the industrial partner
   (a) Persistent data storage should be implemented [15]
   (b) The database management system should be interchangeable [15]
   (c) A Patient entity must be specified with the following fields [17]
      - First Name
      - Last Name
      - Date of Birth
      - Status (Pre-captured, checked, in-treatment, for deletion)
   (d) There should be a one-to-many relation between a Patient and communication data [17]
   (e) Communication data should contain the following [17]
      - Type (phone, mobile, email)
      - Contact (work private relative)
      - Data (phone, mobile, email)
   (f) There should be a one-to-many relation between a Patient and address which contains the following fields [17]
      - Field type (private, work)
      - Street
      - City
3. The system should meet documentation industrial partner requirements
   (a) A description of the deployment process should be provided [15]
   (b) A description of the source code should be provided [15]
   (c) A description of test scenarios and sample testing data should be provided [15]
   (d) The system manual should describe how to easily add or remove fields from the data model [17]
4. Open-source technologies should be used [15]

B System Design

MVP Architecture

The MVP system had the following architecture,

After conducting research, it was decided to use a microservice approach when designing the Patient service. The benefits of microservices are well-known. Scalability is increased since services can be scaled independently, reliability is increased as microservices are isolated from each other, and maintainability is increased because microservices can be developed, deployed, and tested separately [13].

The design choices were based on this microservice approach. *Docker* was used

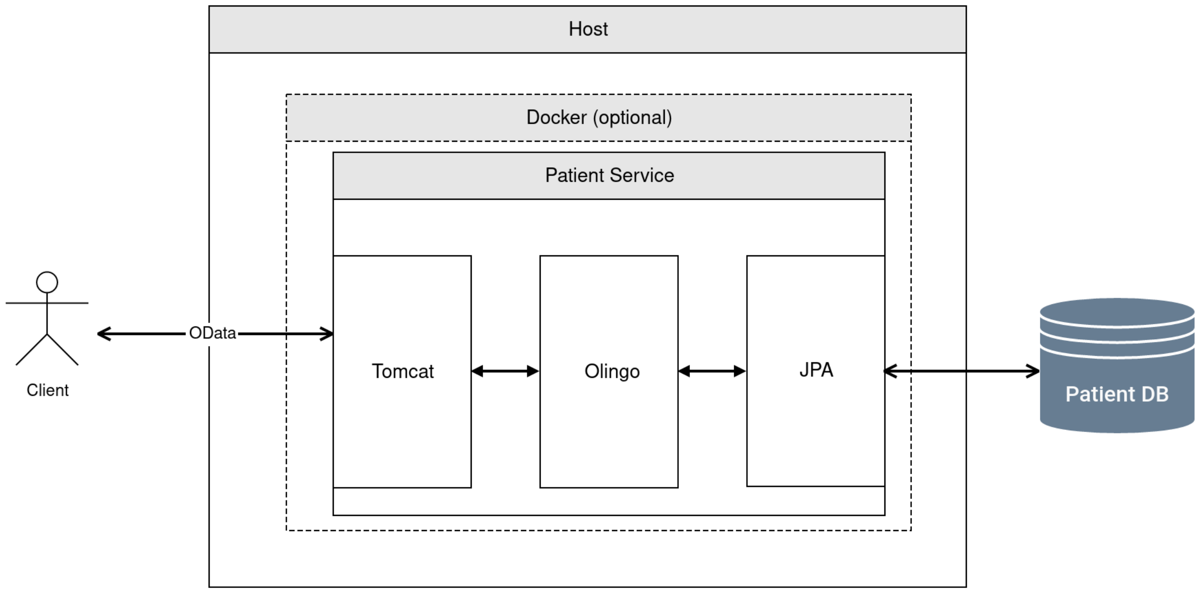

Figure 1: System Architecture for Patient API

to containerise the service and an embedded application server, *Tomcat*, was chosen instead of a traditional, external server. This greatly increased isolation, as everything needed to run the Patient service, aside from the database, was included in the Docker image. Deploying the service was simple as it only required pulling the docker image from docker hub and then running it.

Apache *Olingo* was used to implement OData and *JPA* was used to implement data persistence and allow the database management system to be interchangeable. This made development much easier as only the data model of the Patient service needed to be defined. JPA handled any database operations including creating the database schema from the data model. Olingo created all the API routes automatically using the data model.

Java was chosen as the programming language due to being statically-typed and able to catch errors at compile-time. *Spring-boot* was chosen as the development framework due to offering a wide-range of features, such as configuring the application to work with different databases through defining different profiles. This was used to test the system for database management system independence, using an in-memory H2 database, a local Postgresql database, and a Postgresql file database. Finally *Maven* was chosen to handle dependencies. This proved to be important, as of the time of writing, Olingo supported Javax but not Jakarta [1], so the versions of other modules such as spring-boot and tomcat had to be chosen to be compatible with Javax. Using Maven to specify dependencies meant that the Patient service would not break with future updates of dependencies.

Data model

Compared with the requirements, additional columns were added to provide extra information. Middle name, gender, title were added for Patients, Zip Code, Country, Region, Priority, Description were added for Addresses and Priority was added for Contacts. Initially enums were to be used for look-up fields such as gender, however to increase data normalisation lookup tables were used. This allowed the API to

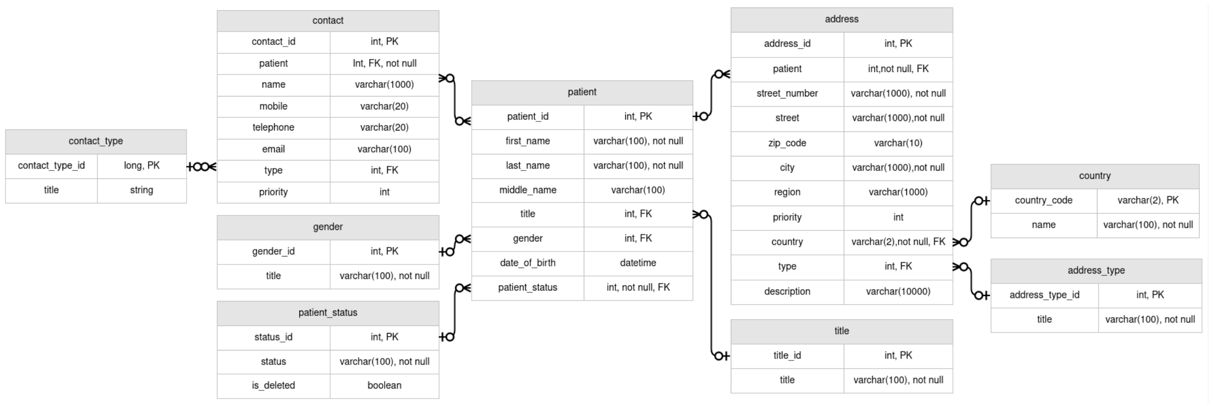
contact
contact_id | int, PK
patient | Int, FK, not null
name | varchar(1000)
mobile | varchar(20)
telephone | varchar(20)
email | varchar(100)
type | int, FK
priority | int
contact_type
contact_type_id | long, PK
title | string
gender
gender_id | int, PK
title | varchar(100), not null
patient_status
status_id | int, PK
status | varchar(100), not null
is_deleted | boolean
patient
patient_id | int, PK
first_name | varchar(100), not null
last_name | varchar(100), not null
middle_name | varchar(100)
title | int, FK
gender | int, FK
date_of_birth | datetime
patient_status | int, not null, FK
address
address_id | int, PK
patient | int,not null, FK
street_number | varchar(1000), not null
street | varchar(1000),not null
zip_code | varchar(10)
city | varchar(1000),not null
region | varchar(1000)
priority | int
country | varchar(2),not null, FK
type | int, FK
description | varchar(10000)
title
title_id | int, PK
title | varchar(100), not null
country
country_code | varchar(2), PK
name | varchar(100), not null
address_type
address_type_id | int, PK
title | varchar(100), not null

Figure 2: ERD of Patient Service

perform CRUD for look-up fields and link look-up fields, e.g. by allowing the patient's gender to be displayed using /Patients(1)/GenderDetails. For key data such as Patients soft-deletes were used, so that when a delete request is sent the API sets the status of a Patient to "deleted" rather than removing the Patient from the database. This allows for auditing of deleted patients.

C Testing

Thorough testing of the Patient service was conducted to ensure it met its requirements. This involved three types of testing: unit testing, integration testing, and performance testing. A public workspace was created in Postman to provide example requests that could be run by cmd or other users wanting to try out the API. Each collection could be run sequentially to populate the database, test for OData compliance by using system queries such as filter, and perform CRUD operations. Initially testing was conducted using the Postman GUI, however testing was later switched to coded test scripts to provide more advanced and customisable testing.

Testing best practices were followed, with a separate test directory created and test scripts written for each class in the data model, e.g. PatientTest.java. *JUnit* was used for unit testing. Unit testing focused on testing the get and set methods of every class in the data model. The input provided to the set method was compared with the output obtained using get after set was called. If the get and set methods were working correctly the two values should be equal. This type of testing was quite basic but ensured that all the fields in the data model could be get and set correctly.

For integration testing, application-test.properties was used to create a database every time testing was conducted that would be deleted after testing was complete. @SpringBootTest was used to start tomcat listening on port 8079 for test HTTP requests. Integration testing involved testing CRUD POST, GET, PATCH/PUT, DELETE requests for every entity in the data model. Negative testing was also used with invalid data such as invalid foreign keys, or input failing validation such as being too short or the wrong type. For invalid write requests, the entity in the database was tested to ensure it was unchanged. For valid write requests, the entity in the database was tested to ensure the created/updated entity matched the request body JSON. Fields not included in the request body were tested to ensure they were unchanged. Delete requests were tested to ensure that the entity was not visible after being deleted. Both unit and integration testing could be run using mvn clean test

Gatling was used for performance-testing due to being lightweight and allowing for test scripts to be written in Java. This meant that peformance testing could be run using mvn gatling:test. Up to 750 users were simulated arriving at a rate of 50 users per second to simulate real-usage where users could be arriving gradually. Each user would add a patient, add an address for that patient, view the added patient and address. Random inputs were used to simulate real-world usage.

4 Extension Methodology

A Requirements

5. The system must be compliant with relevant GDPR principles [11]

   (a) Must comply with principle (a) Lawfulness, fairness, and transparency, specifically transparency by allowing the data subject (patient) the ability to be informed on who has access to their data.

   (b) Must comply with principle (c) data minimisation. In the context of this project, this is limiting access to medical records to what is necessary for an employee's role.

   (c) Must comply with principle (d) accuracy by allowing the data subject to be given the ability to update certain records

   (d) Must comply with principle (f) integrity and confidentiality. This will involve implementing integrity by recording changes made to data and enforcing confidentiality by ensuring data can only be written and/or read if the user has sufficient authorisation.

   (e) Must enforce confidentiality i.e. data can only be read or written if the user has sufficient authorisation.

   (f) Must enforce integrity by recording changes made to data.

6. The system must meet the requirements set-out by the industrial partner

   (a) The system should support the following types of users [17]

   - Patient (referred to as Login by cmd) - read all identical identities in status Checked or In-treatment
   - Admin - can perform any action in the system
   - Assistance - change all fields for patients with a status of parked or for deletion. Otherwise only change communication and addresses

   (b) Status should only be changed by OData function e.g. Patients?setStatus(). This is referred to as function exports by cmd [17]

   (c) API requests should be logged [17]

   (d) API errors should be logged [17]

B System Design

When designing the extension, an initial problem was determining whether to implement permissions in the database or at the code-level using Spring Security.

When using code-based permissions, the database only needs to store roles and employee mapping for roles 3. When sending requests, first the user is authenticated and provided with an access token, e.g. a JSON Web Token (JWT). Using the IAM service, the JWT can include a list of roles associated with the user. When the user sends a request to the Patient service, the get and set methods in the data model are secured with @Secured(ROLE) to control which roles have access to which methods and implement the permissions.

When using database-based permissions, the database also needs to store a role's mapping to entities and fields in the system 5. For read-authorisation, after authentication, the gateway gets the fields a user has access too by sending a request to the IAM service /Authorisation(entity=..&emp=..). The list of fields returned can then be used to only show certain fields to the user by appending $select to the read request. If there are conditions to when a record can be read, these are stored in the database as the filter_expression and can be appended to the read request using $filter.

$filter and $select cannot be sent with a write request so authorisation is slightly different. After authorisation and receiving the list of allowed fields to be written to, the write request body is compared with the list of fields the user can update. If the user is attempting to write to any unauthorised fields, the user's request is reject and a HTTP error can be raised. Otherwise, if the user can only update certain records a /Patients(..)?$filter.. is sent to see if this record exists. If it does, the update request can be sent, otherwise a HTTP error can be raised.

The following table compares the two approaches using different criteria.

Criterion	Code-based permissions	Data-based permissions
Performance: code-based	1 round-trip required for all requests	3 round-trips required for write requests and parsing of request body. 2 round-trips required for read requests
Transparency: data-based	Can see what entities a role has access to using code and what role a user has using the IAM service	Can send requests to the IAM service to see exactly what entities and fields a user has access to
Maintainability: code-based	All code for data model (functionality, authorisation, validation) in the same place	Significantly more complex IAM database. IAM database is decoupled from the Patient data model, so if a field name is changed in the data model but not the database, authorisation can be broken.
Complexity: code-based	Only requires IAM database	Requires IAM service and gateway

It was decided to proceed with the code-based approach due to winning 3 out of the 4 criteria. The criteria it performed worse in, transparency, could be mitigated by allowing the user to request a log of all users who have previously accessed their records.

5 Results

The developed Patient service met the requirements set out. Interoperability 1 was achieved using Olingo, a data model 2 was defined and implemented using JPA, Documentation 3 was added to separate pages on the project's GitHub, and throughout the design process open source technologies were used 4.

Thorough testing provided an objective way of verifying that the developed system met the requriements. In particular, using Gatling it was discovered the system could support over 750 concurrent users, far more than the 50 users needed by the industrial partner.

A discussion meeting was held with the industrial company where the industrial contact saw a demonstration of the system and is currently signing off on the initial patient service.

The system also exceeded the initial requirements in several aspects. Firstly, a microservice approach was implemented during the design and development phase, allowing the system to have numerous benefits such as increased scalability, better fault tolerance and increased maintainability. Secondly, extensive testing was incorporated to analyse the system's performance and find bugs.

Finally, detailed documentation was written on GitHub including a Javadoc to ensure there is extensive support for users and developers of the service. This also represented an extension for this project compared with existing solutions, as existing solutions do not go into detail on how to test the system.

6 Discussion

The first key objective of interoperability is met by the Patient service. The second key objective of ensuring the system is GDPR-compliant is planned to be implemented in the second release of the healthcare management system.

7 Future Work

One of the main ways the Patient service could be improved is through a focus on performance. Currently the spring-boot servlet stack is used which is blocking and multithreaded. Switching to a reactive stack which is non-blocking, event driven, and asynchronous could increase performance by 5x depending on the load [14].

References

[1] Apache. Interface odatajpacontext. 2022. URL https://olingo.apache.org/javadoc/odata2/org/apache/olingo/odata2/jpa/processor/api/ODataJPAContext.html.

[2] Smile CDR. Hapi fhir documentation. 2022. URL https://hapifhir.io/hapi-fhir/docs/.

[3] OneTrust DataGuidance. Sweden: Datainspektionen completes audit of capio st. göran's hospital, imposes fine of sek 30m. 2020. URL https://www.dataguidance.com/news/sweden-datainspektionen%C2%A0completes-audit-capio-st-g%C3%B6rans.

[4] Karine pontet Alexandra Georges Deloitte: Luc Brucher, Kim Mosel. A jour-

ney towards smart health. the impact of digitalization on patient experience. 2018. URL https://www2.deloitte.com/content/dam/Deloitte/lu/Documents/life-sciences-health-care/lu_journey-smart-health-digitalisation.pdf.

[5] NHS Digital. Personal demographics service - fhir api. . URL https://digital.nhs.uk/developer/api-catalogue/personal-demographics-service-fhir#api-description__security-and-authorisation.

[6] NHS Digital. National role-based access control (rbac) for developers. . URL https://digital.nhs.uk/developer/guides-and-documentation/security-and-authorisation/national-rbac-for-developers.

[7] HL7). Fhir release 4b. URL https://www.hl7.org/fhir/.

[8] Healthcare Information and Management Systems Society (HIMSS). Interoperability in healthcare. URL https://www.himss.org/resources/interoperability-healthcare.

[9] OASIS). Odata version 4.01. 2020. URL https://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part1-protocol.html.

[10] National Institute of Standards and Technology (NIST). Attribute based access control (abac). URL https://csrc.nist.gov/Projects/Attribute-Based-Access-Control.

[11] Information Comissioner's Officer. Guide to the uk general data protection regulation (uk gdpr). URL https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/.

[12] Phillips). Future health index. 2016. URL https://www.himss.org/resources/interoperability-healthcare.

[13] Chris Richardson. *Microservices Patterns*. Manning, 2018.

[14] Raj Saxena. Springboot 2 performance — servlet stack vs webflux reactive stack. 2018. URL https://medium.com/@the.raj.saxena/springboot-2-performance-servlet-stack-vs-webflux-reactive-stack-528ad5e9dadc.

[15] Franz-Josef Ziegert Stefan Poslad. Industrial partner project brief. 2021. URL https://www.overleaf.com/project/62a0d1b65ec6c0593e7c477e/file/62a642f26b3b3bd50ecd72ae.

[16] Ben Wolford. What is gdpr, the eu's new data protection law? URL https://gdpr.eu/what-is-gdpr/.

[17] Franz-Josef Ziegert. Industrial partner minimum viable product survey response. 2022. URL https://www.overleaf.com/project/62a0d1b65ec6c0593e7c477e/file/62a645656b3b3b678ecdcd5e.

8 Appendix

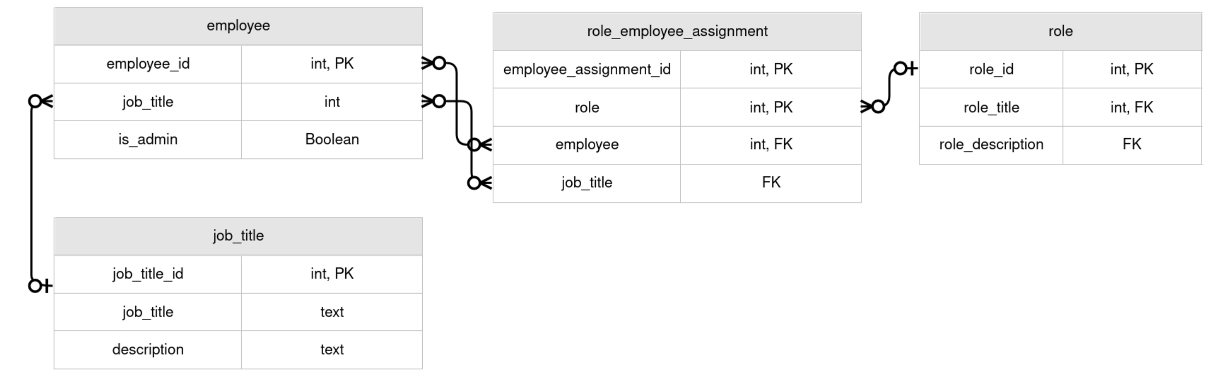

Figure 3: ERD for extension using code-based permissions

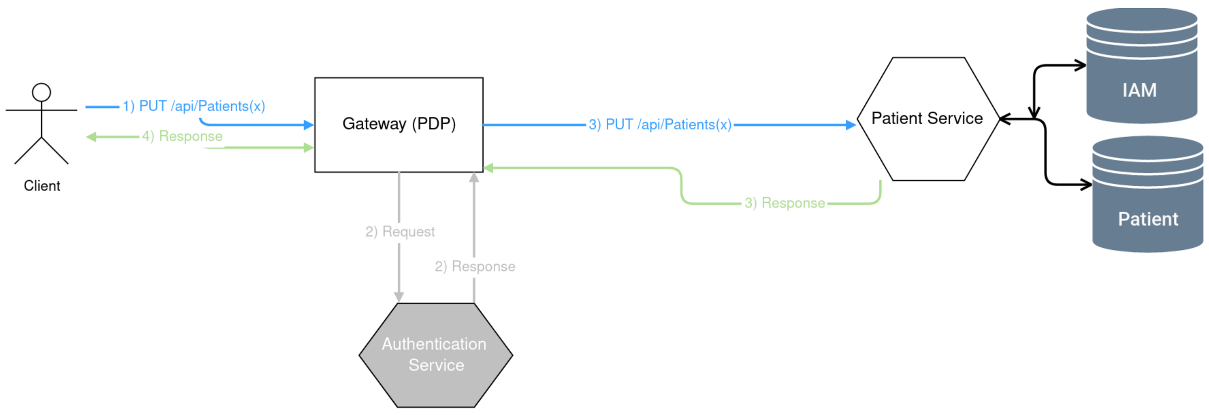

Figure 4: Authorisation flow for extension using code-based permissions

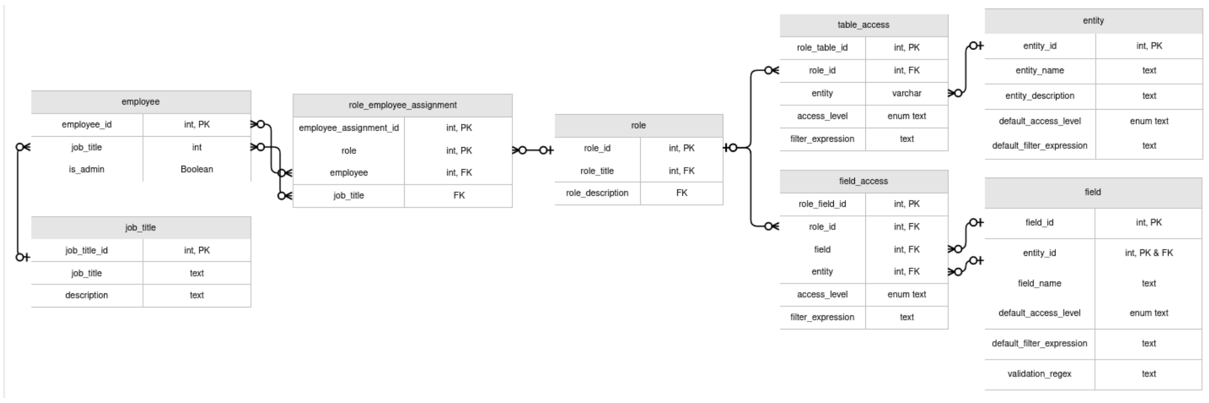

Figure 5: ERD for extension using data-based permissions

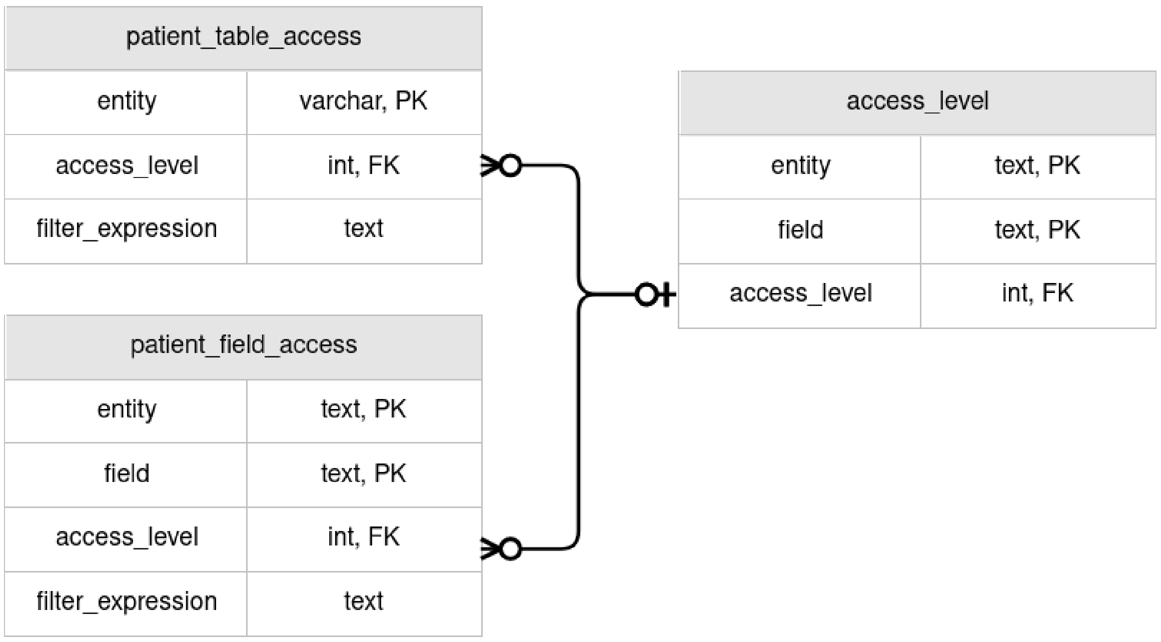

Figure 6: ERD for patient RBAC for extension using data-based permissions

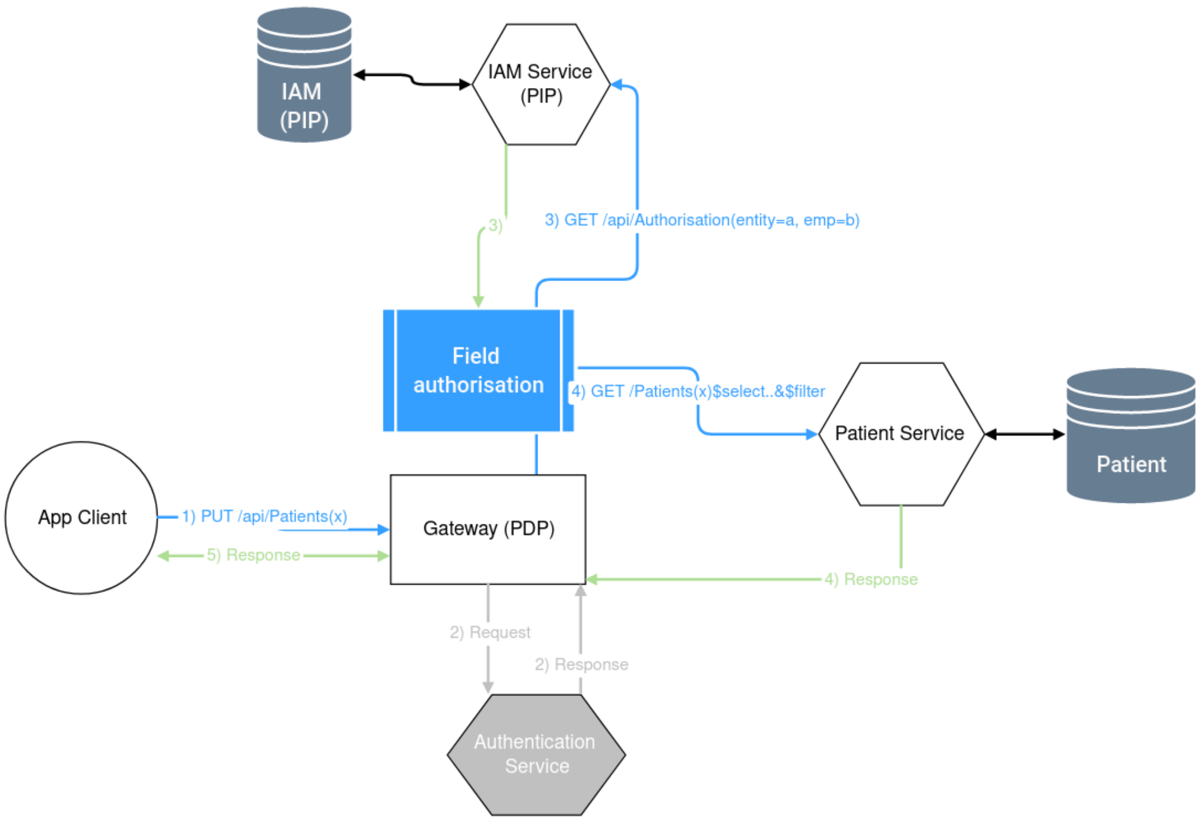

Figure 7: Read authorisation flow for extension using data-based permissions

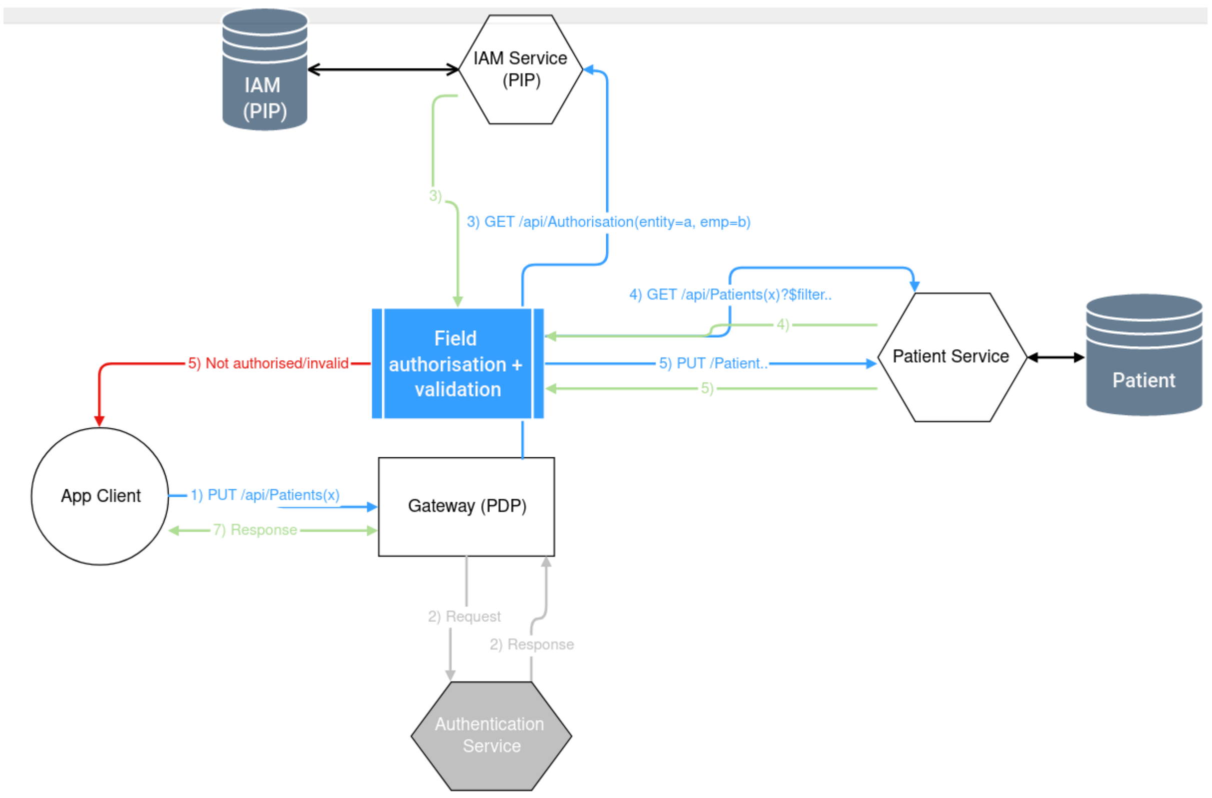

Figure 8: Write authorisation flow for extension using data-based permissions