Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 160 (2019) 338–345

Procedia Computer Science

www.elsevier.com/locate/procedia

The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2019)
November 4-7, 2019, Coimbra, Portugal

A GraphQL approach to Healthcare Information Exchange with HL7 FHIR

Suresh Kumar Mukhiya[a,*], Fazle Rabbi[a,b], Violet Ka I Pun[a,c], Adrian Rutle[a], Yngve Lamo[a]

[a] *Western Norway University of Applied Sciences, Bergen, Norway*
[b] *University of Bergen, Norway*
[c] *University of Oslo, Norway*

Abstract

Interoperability is accepted as a fundamental necessity for the successful realization of Healthcare Information Systems. It can be achieved by utilizing consistent standards defining syntactic and semantic meaning of the information being exchanged. HL7 FHIR is one of such open standards for Health Information Exchange (HIE). While HL7 FHIR supports Representational State Transfer (REST) architecture and Service-oriented Architecture (SOA) for seamless information exchange, it inherits the inflexibility and complexity associated with the RESTful approach. GraphQL is a query language developed by Facebook that provides promising techniques to overcome these issues. In this paper, we exploit the use of GraphQL and HL7 FHIR for HIE; present an algorithm to map HL7 FHIR resources to a GraphQL schema, and created a prototype implementation of the approach and compare it with a RESTful approach. Our experimental results indicate that the combination of GraphQL and HL7 FHIR-based web APIs for HIE is performant, cost-effective, scalable and flexible to meet web and mobile clients requirements.

Keywords: GraphQL; HL7 FHIR; REST API; overfetching; underfetching; Health Information Exchange; Interoperability; REST vs GraphQL

1. Introduction

Software interoperability in the healthcare domain [10] can be realized by utilizing consistent standards like HL7 FHIR that supports SOA and RESTful approach to seamless information exchange. The RESTful architecture enables a machine to machine communication using only the ubiquitous HTTP protocol without additional layers. In addition, REST API [7] is a standard for deploying Application Programming Interfaces (APIs) for both simple and complex

* Corresponding author. Tel.: +47-94430044
E-mail address: skmu@hvl.no

1877-0509
10.1016/j.procs.2019.11.082

web applications. REST provides a comprehensive set of rules and constraints that can deliver fully functioning web services and structured access to resources [17]. However, these rules and constraints become inflexible and complex due to various reasons: i) the increase in the complexity of the systems being built, ii) the demand in the higher quality of services by the system end users, iii) the development of highly efficient real-time systems, and iv) the dynamic data fetching requirements of the mobile and web clients. To mitigate this inflexibility and complexity associated with REST, a new standard, known as GraphQL [5], has been developed by Facebook. Specifically, the following issues that are associated with REST are addressed by GraphQL:

- **Query Complexity**: REST requires multiple and complex HTTP requests to fetch multiple resources.
- **Overfetching**: Overfetching is characterized by returning more data than required by an application.
- **Under-fetching and n+1 request problem**: Under-fetching indicates that a particular endpoint does not give sufficient information. This results in making an additional request by a client application to a server. This is referred to as n+1 request problem.
- **API versioning**: An API creates a contract between two systems for information exchange and hence it should be stable and consistent. However, the business goals of any organizations change, so the APIs must be adaptable for modifications according to their behavior. This is handled by API versioning. An empirical study [12] addresses the perennial issue of REST API versioning and how evolution of such API affects the clients.

GraphQL has the potential to overcome these issues as it can be used to create scalable, sustainable, flexible, maintainable, interoperable, and secured APIs [11]. GraphQL functions as a service abstraction layer [16] providing a single API endpoint for resource fetching, creation or modification. GraphQL APIs holds the following characteristics: i) strongly typed schema; ii) introspection that allows clients to query about fields, types and supported queries; iii) enabling rapid product development; iv) rich open-source ecosystem; v) composable API (schema federation which allows merging multiple GraphQL APIs) [5]; vi) faster request-response cycles; vii) client-specified queries; and viii) being hierarchical (a GraphQL query itself is structured hierarchically and is shaped just like the data it returns) [5]. The *introspection features* provided by GraphQL allows users and developers to comprehend the interface easily and *machine-readable representation* enables dynamic and loose coupling between server and clients.

Despite these features, to the best of our knowledge, GraphQL based APIs are not adapted for exchanging healthcare information in general and especially not using *HL7 FHIR*. In this paper, we present a quantitative constructive research to evaluate the applicability of GraphQL based APIs for HIE based on HL7 FHIR [4]. We also propose an algorithm to automatically produce GraphQL schema for HL7 FHIR resources. The schema generation is a model transformation approach which reduces the number of errors typically occurring at the time of schema development.

The rest of the paper is structured as follows: Section 2 provides a mapping of existing HL7 FHIR resources to GraphQL schema. Section 3 describes the prototype implementation from healthcare context. Section 4 explains evaluation criteria and results. Section 5 discusses related works, existing challenges and concluding remarks.

2. Mapping HL7 FHIR Resources to GraphQL Schema

HL7 FHIR is based on *Resources* which are the common building blocks for all information exchanges. A single resource (e.g., Patient) contains several *Element Definitions* (e.g., name) which has *Data Type* (e.g. String) and cardinality associated with it (Figure 1). DataType can be Scalar Type, Enumeration Type or other HL7 FHIR resource types. Moreover, *Element Definition* can also reference another HL7 FHIR resource. GraphQL supports Interface Definition Language (IDL) that defines schemaes. A GraphQL document [5] can contain one to many schemas. As shown in Figure 1, each schema contains at least one RootOperationTypeDefinition (e.g., query) and several Types definition. Each Type definition can have several Fields which may contain Name(picture), Arguments(size) and Type(URL) definition. Based on the transformation model illustrated in Figure 1, we present an algorithm to transform from HL7 FHIR resources to GraphQL schema. In Algorithm 1, the recursive function recursive_hl7fhir_graphql_mapper takes the HL7 FHIR resource as an input and returns a GraphQL schema as an output. The function iterates over each field in the HL7 FHIR resource and based on field type and cardinality, an equivalent schema is generated as follows:

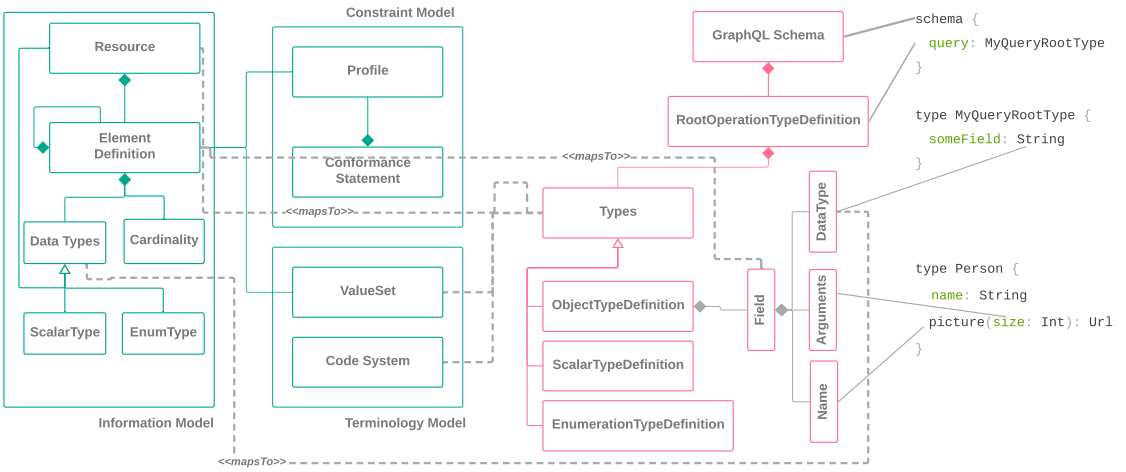


Fig. 1: A HL7 FHIR resource contains *information model, constraint model and terminology model*. Each field (Element Definition) in HL7 FHIR resources is mapped to an equivalent Type in GraphQL schema

Algorithm 1: Mapping HL7 FHIR resources to GraphQL schema

Input: FHIR Resource: Element Definition(field), Data Type(type)
Output: GraphQL Schema

```
function recursive_hl7fhir_graphql_mapper (Resource)
 schema = {};
foreach field ∈ HL7 Resource.fields do
switch Resource.field do
    case field.Type is ScalarTypeDefinition do
        if field.cardinality is 0,1 then
            add_to_schema(field, type)
        end if
        if field.cardinality is 0,* then
            add_to_schema_as_list(field, type)
        end if
    end case
    case field.Type is EnumTypeDefinition do
        if EnumTypeDefinition already exists then
            - reference to schema
        else
            - define_new_type_enum(**args)
            - reference to schema
        end if
    end case
    case field.Type is Custom OR field.Type is HL7 FHIR
      Resource do
        if Custom OR Resource already exists then
            - reference to schema
        else
            - define new type Resource
            - reference to schema
            - recursive_hl7fhir_graphql_mapper(Resource)
        end if
    end case
end switch
;
return schema
```

```
{
  "family" : "<string>",
  "given" : ["<string>"],
  "use" : "<code>", // usual | official | temp |
        nickname | anonymous | old | maiden,
  "period" : { Period }
}
```

Listing 1: Patient Name in HL7 FHIR format

- If the field type is ScalarTypeDefinition (String, Float, Int, Boolean, ID) and cardinality is either 0 or 1, we simply add to the schema with same field name and type. If the cardinality of field is 0 to *, we add to the schema as ListType. For example Listing 1 shows a patient name in HL7 FHIR format. The field family has data type string, so the field is simply added to GraphQL schema with the same name and data type. The field given is an array of datatype string. Hence, it is added as ListType in the GraphQL schema.
- If the field type is EnumTypeDefinition, and if it is already defined, we add field to schema and reference it to the field. If it is not defined, we create a new schema of enum type with required arguments and reference it to the field. A generated schema for the field use in Listing 1 can be found in Listing 2.
- If the field type is HL7 Resource or Custom, and if it is already defined in the schema, we reference it to that field. If it is not yet defined, a recursive call to recursive_hl7fhir_graphql_mapper is made with this field as the argument.

3. Prototype Implementation

This section outlines a typical health information exchange scenario in the context of healthcare. The scenario is taken from an ongoing project INTROMAT [9], which aims to support mentally ill patients with adaptive technologies. We discuss the need for GraphQL based API with respect to a self assessment mobile application, and to INTROMAT core architecture [14] that consumes GraphQL-based API for HIE.

3.1. INTROMAT Core Architecture

As illustrated in Figure 2, INTROMAT core architecture contains the following main components communicating over SOA standards:

- **Authorization Server** is an OpenID connect [15] compliant web server with an ability to authenticate users and grant authorization access tokens. Moreover, the authorization server manages scopes and permission of the clients, introspects token and requests for the resource server.
- **Resource Server** is a FHIR [1] compliant web server with an ability to respond to HTTP requests consuming access tokens granted by the Authorization Server.
- **Web Client** provides an interface for therapist and administrators to login and view the list of patients, questionnaires and other resources.
- **Mobile Client** is a self-assessment mobile application (Section 3.2) that consumes Questionnaire HL7 FHIR resources and sends response by using QuestionnaireResponse resource.

```
enum enumNameType {
   official
   usual
   temp
   nickname
   anonymous
   old
   maiden
}

type Period {
  start: Date,
  end: Date
}

type HumanName {
   family: String
   given: [String]
   use: enumNameType
   period: [Period]
}
```

Listing 2: Generated GraphQL schema from Listing 1

A detailed technical documentation of the prototype [13] of the above architecture is available to interested readers. All the components of the prototype are hosted on Amazon Web Server[1] (AWS) on t3.micro (EU, Stockholm) Red Hat Linux instances to perform the evaluation under similar environment.

3.2. Self Assessment Mobile Application

The main aim of developing the self-assessment application is to provide a possible way for people suffering from mental health morbidity to self-evaluate and manage their illness. In addition to, the application showcases the exchange of information based on HL7 FHIR standard [14] to support interoperability. The application utilizes the REST API standard to exchange information between *mobile client* and *resource server*. The application contains several views including: i) a list of available self-assessment questionnaires (name, id), and ii) a detailed view of a questionnaire (id, questions(id, text), and options (id, text)). The listing view (all available questionnaires) shows the name of the questionnaire and only requires to have id, name of the questionnaire. However, to support interoperability and maintain the semantics, we require to support all the available attributes[2] of the questionnaire resource mentioned in the HL7 FHIR

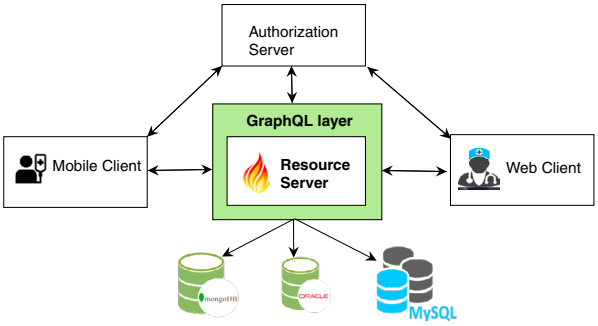


Fig. 2: The prototype contains five major components: a) mobile client for self-assessment, b) resource server based on HL7 FHIR, c) authorization server for authentication and authorization, d) web client: to provide web interface for therapist and e) mongoDB to store data.

[1] https://aws.amazon.com
[2] http://hl7.org/fhir/questionnaire.html#resource

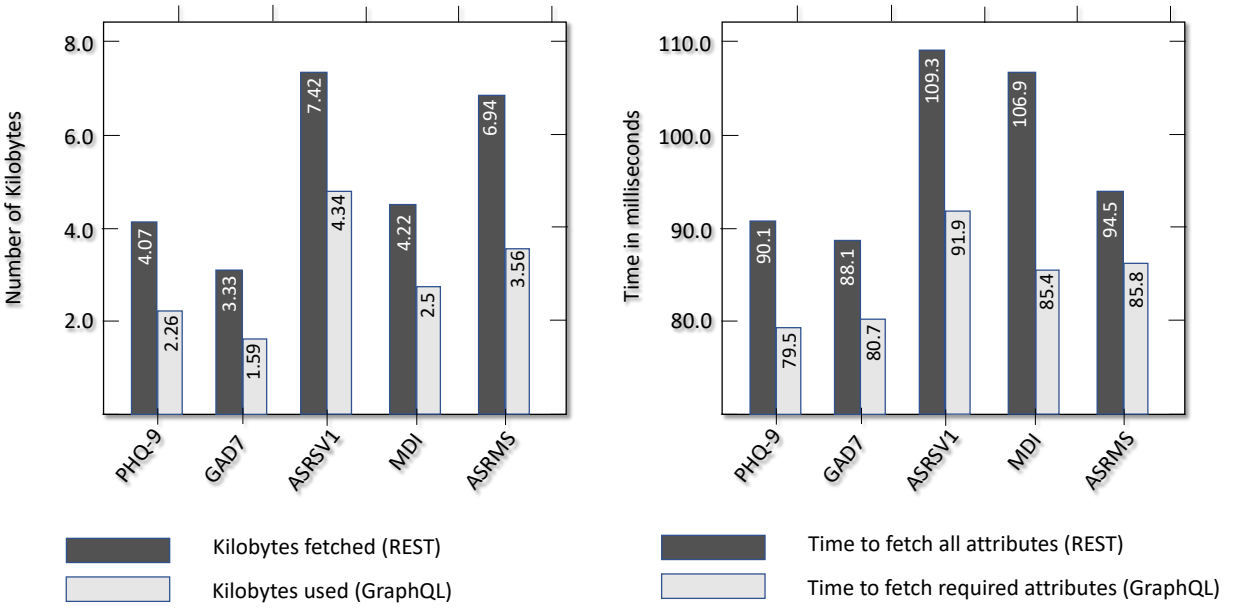


Fig. 3: (Left:) Response size – *Questionnaire:* types of self-assessment questionnaire for mental health, *Kilobytes Fetched:* bytes of data fetched by REST API, *Kilobytes Used:* data actually used by mobile client, (Right:) Response Time – *All attributes*: Time taken to fetch all attributes, *Required attributes*: Time taken to fetch used attributes

[3] standard. These API endpoints fetch a plethora of additional metadata that are irrelevant for the application. The `RESTful` approach to solving this problem is to define a new API endpoint or update an existing endpoint that would only return `id` and `name` of the questionnaire list. However, creating new endpoints or modifying existing endpoints for solving different requirements of the clients become quickly inflexible and expensive. This is because different clients require different attributes for different views and these requirements are very dynamic which are liable to change according to demographics, organizational ethics, and application views. This can be solved by providing an endpoint with a higher level of abstraction for clients to query the server based on their requirements. The need for such endpoints is supported by the empirical study [12].

4. Evaluation

We aim to evaluate whether migrating from `RESTful` approach to `GraphQL` based API is scalable and performant. To evaluate, we performed *response size/time test* and *performance test*. As aforementioned, all the components are hosted at Amazon Web Server with the same configuration for testing to keep evaluation metrics consistent.

4.1. Response size and time

The aim of this evaluation is to explore how much extra information was fetched from the endpoints when fetching a single `Questionnaire` item using REST and how much is actually being used by our self-assessment mobile application (see Section 3). Figure 3 (left) illustrates the overall difference in the amount of data returned by `HTTP` responses while fetching a single questionnaire resource. The figure shows that approximately 50 percent of information is not used by our self assessment mobile application. We also evaluated the time taken to fetch all the attributes in `RESTful` approach and compare the result with the time taken to fetch only used information using GraphQL API. Figure 3 (right) shows the time taken in milliseconds to fetch all the attributes versus the time taken to fetch only the

required attributes for various types of questionnaire. To keep the measurement uniform, Postman[3] was used to send HTTP requests to the server; the evaluation was performed on the same machine and we took an average reading (10 requests were recorded for each questionnaire type).

4.2. Performance Testing

Performance testing inspects responsiveness, stability, scalability, reliability, speed and resource usage of software and infrastructure. We used BlazeMeter[4] which is powered by Apache JMeter[5] for creating performance tests. Each HL7 FHIR resource requires the endpoints for CRUD (Create, Read, Update, Delete). Presenting performance tests for each endpoint is out of the scope of the paper. However, we present performance tests for getting an Questionnaire resource, where the questionnaire GAD-7 is for anxiety disorder. Table 1 shows the metadata for the performance evaluation, which include two parts: one for fetching all the attributes from questionnaire, and one for fetching only required attributes). Both were performed on the same server and have the same configurations. Fifty virtual users requested resources concurrently for 20 minutes. Based on this test, we have made the following observations:

Description	All attributes	Required Attributes
Average Throughput	100.6 hits/second	157.7 hits/second
Average Response Time	484 millisecond	308 millisecond
Time Elapsed	20 minutes	20 minutes
Concurrent Users	50	50

Table 1: Performance test meta-data for fetching GAD-7 Questionnaire resource. Column 1: description of the meta data, column 2: meta data for fetching all attributes from the endpoints. column 3: meta data when fetching only required attributes

- Figure 4a illustrates concurrent users versus average throughput[6] (Hits/s) when requesting all the attributes using RESTful approach from GAD-7 Questionnaire.
- Figure 4b illustrates concurrent users versus average response time[7] when requesting all the attributes using RESTful approach .
- Figure 5a illustrates concurrent users versus average throughput (Hits/s) when requesting only required attributes using GraphQL approach.
- Figure 5b illustrates concurrent users versus average response time when requesting only required attributes using GraphQL approach.

Figures 5a and 5b clearly show that there is an increase in average throughput and that response time is faster if only the required attributes are fetched by using the GraphQL based API. The result is interesting to related stakeholders as the throughput and response time are directly associated with operating costs and performance of the system, which in turn are associated with the user adherence.

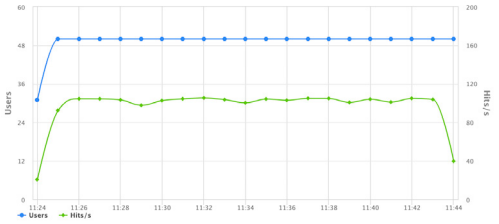
4.3. Expert Evaluation

The code for all the components mentioned in Section 3.1 has been evaluated by three senior front-end developers to check their compliance with ISO/IEC 25000:2005 software product quality requirements and evaluation [11] criteria and any presence of anti-patterns [2].

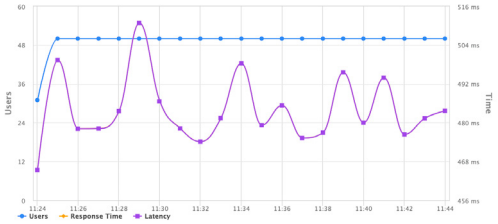
4.4. Discussion

Although GraphQL provides a sophisticated technology to develop client-centric applications with very complex queries, there are some challenges. For example, unmanaged queries can have security implications. A malicious

[3] https://www.getpostman.com/
[4] www.blazemeter.com
[5] https://jmeter.apache.org/
[6] average number of HTTP/s requests per second generated by the test
[7] average amount of time from first bit sent to the network card to the last byte received by the client.

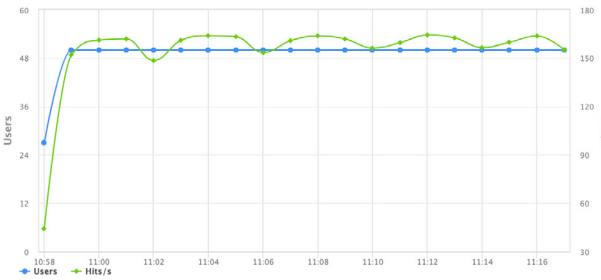


(a) Concurrent users and number of hits per second when fetching all the available attributes from the `Questionnaire`

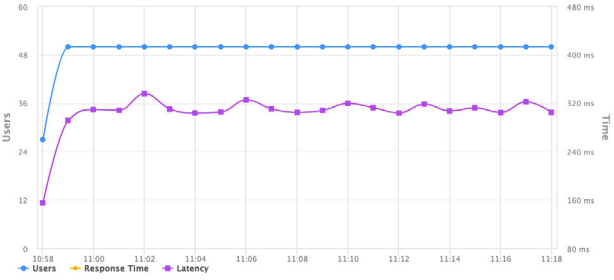


(b) Concurrent users and response time (milliseconds) when fetching only the required attributes from the `Questionnaire`

Fig. 4: Performance test results representing concurrent access using `RESTFul` approach



(a) Concurrent users and number of hits per second when fetching only the required attributes from the `Questionnaire`



(b) Concurrent users and response time (milliseconds) when fetching only the required attributes from the `Questionnaire`

Fig. 5: Performance test results representing concurrent access using `GraphQL` approach

actor could submit an expensive, nested query to overload a `GraphQL` server, database, and network. The absence of appropriate protections can open up to a `DoS` attack [6]. Another challenge is to deal with schema duplication. The creation of `GraphQL` based backend, which acts as a database service abstraction layer, involves a plethora of similar but not-quite-identical code, especially schema definition. It requires i) a schema definition based on the choice of the database being used (`MongoDB` is used in this paper, and therefore schemaes are based on `Mongoose`[8]); and ii) a schema definition for a `GraphQL` endpoint. This creates schema redundancy and requires synchronization between two independent sources of truth.

5. Related Work

There is a number of emerging solutions in the `GraphQL` echo-system including `PostGraphile`[9] that generates `GraphQL` schema from `PostgreSQL` database, and `Prisma`[10] that allows generating queries and mutations. In addition to this, various other transformation libraries are being developed by the community to support various database systems. There has been research on the syntax and semantic representation of `GraphQL`. A recent study by Ulrich et. al [16] introduces *QL*^{4}*MDR*, an ISO 11179-3 compatible `GraphQL` query language, which promotes interoperability by defining a uniform interface between different metadata repository (MDR) allowing querying over a distributed network. However, their study does not explicitly give the answer to how `HL7 FHIR` can be used for HIE and how `HL7 FHIR` resources can be transformed into the `GraphQL` schema. Our work helps bridge this gap and is different from the work in [16] as we focus on experimental evaluation to demonstrate the applicability of `HL7 FHIR` and `GraphQL` based API in a healthcare setup. Another study was presented in [8] which formalizes the semantics of `GraphQL` queries

[8] https://mongoosejs.com/
[9] https://www.graphile.org/postgraphile/
[10] https://www.prisma.io/

based on the labeled-graph data model and analyzes the language to demonstrate that it admits efficient evaluation methods. Moreover, the study proves that the complexity of GraphQL evaluation problem is NL-complete and the enumeration problem can be solved with constant delay.

6. Conclusion

Interoperability in healthcare information system can be achieved by using standards like HL7 FHIR which supports RESTful and SOA approaches by default. However, the RESTful approach comes with certain shortcomings. We have summarized a list of challenges with the RESTful approach in HIE and presented a GraphQL based API for HIE using HL7 FHIR standard as the solution to those challenges. Moreover, we have presented a transformation algorithm that takes HL7 FHIR resource definition as input and produces GraphQL schema as output. Furthermore, we present a healthcare application (self-assessment mobile application) based on a reference architecture proposed in [14]. The evaluation of the healthcare application shows that the use of a GraphQL based API is both performant and cost-effective. In addition, it solves problems associated with the RESTful approach, including the over-fetching, under-fetching and, n+1 request. The most prominent future work involves removal of schema duplication by using transformation tools, and creation of comprehensible dashboard for better visualization for therapists.

Acknowledgement

This publication is a part of the INTROducing Mental health through Adaptive Technology (INTROMAT) project, funded by Norwegian Research Council (259293/o70). INTROMAT is a research and development project in Norway that employs adaptive technology for confronting these issue. The paper is also partially supported by SIRIUS: Centre for Scalable Data Access (www.sirius-labs.no).

References

[1] Hl7 FHIR SMART app launch, Retrieved December 28.
[2] Ambler, S. *Process Patterns Building Large-Scale Systems Using Object Technology*. 1998.
[3] Bender, D., and Sartipi, K. HL7 FHIR: An agile and RESTful approach to healthcare information exchange. In *Proceedings of CBMS 2013 - 26th IEEE International Symposium on Computer-Based Medical Systems* (2013).
[4] Bryant, M. Graphql for archival metadata: An overview of the ehri graphql api. In *2017 IEEE International Conference on Big Data (Big Data)* (Dec 2017), pp. 2225–2230.
[5] Facebook. Graphql: A query language for apis, Retrieved April 11, 2019.
[6] Ferguson, P., and Senie, D. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. Tech. rep., 2000.
[7] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000.
[8] Hartig, O., and Pérez, J. Semantics and Complexity of GraphQL .
[9] INTROMAT. INTROducing Mental health through Adaptive Technology. https://intromat.no/, 2019.
[10] Iroju, O., Soriyan, A., Gambo, I., and Olaleke, J. Interoperability in Healthcare : Benefits , Challenges and Resolutions. *International Journal of Innovation and Applied Studies ISSN* (2013).
[11] ISO/IEC 25000. ISO/IEC 25000:2005 Software Engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE, 2005.
[12] Li, J., Xiong, Y., Liu, X., and Zhang, L. How does web service API evolution affect clients? In *Proceedings - IEEE 20th International Conference on Web Services, ICWS 2013* (2013).
[13] Mukhiya, S. K., and Rabbi, F. A GraphQL approach for Healthcare Information Exchange with HL7 FHIR: Extended Version. https://github.com/sureshHARDIYA/phd-resources/tree/master/Papers/GraphQLHIE, 2019.
[14] Mukhiya, S. K., Rabbi, F., Pun, K. I., and Lamo, Y. An architectural design for self-reporting e-health systems. In *Proceedings of the 1st International Workshop on Software Engineering for Healthcare* (Piscataway, NJ, USA, 2019), SEH '19, IEEE Press, pp. 1–8.
[15] Sakimura, N., Bradley, J., B. Jones, M., Medeiros, B., and Mortimore, C. Final: OpenID Connect Core 1.0 incorporating, 2014.
[16] Ulrich, H., Kern, J., Tas, D., Kock-Schoppenhauer, A. K., Ückert, F., Ingenerf, J., and Lablans, M. QL 4 MDR: A GraphQL query language for ISO 11179-based metadata repositories. *BMC Medical Informatics and Decision Making* (2019).
[17] Vogel, M., Weber, S., and Zirpins, C. Experiences on Migrating RESTful Web Services to GraphQL. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2018).