2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS)

978-1-6654-6387-4

Web Service Modeling for GraphQL Based College Data Service Access

Irfan Darmawan
Department of Information System
Telkom University
Bandung, Indonesia
irfandarmawan@telkomuniversity.ac.id

Alam Rahmatulloh
Department of Informatics
Siliwangi University
Tasikmalaya, Indonesia
alam@unsil.ac.id

Rohmat Gunawan
Department of Informatics
Siliwangi University
Tasikmalaya, Indonesia
rohmatgunawan@unsil.ac.id

2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS) | 978-1-6654-6387-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICADEIS56544.2022.10037508

***Abstract*—The Higher Education Database or Pangkalan Data Perguruan Tinggi (PDDIKTI) is a collection of data on the implementation of higher education that is nationally integrated. The data contained in the academic information system of a university must be entered in PDDIKTI. The PDDIKTI Feeder application is one of the services that can be used by every university to help facilitate data input. The large and varied data on the implementation of higher education is an obstacle in the data input process into the PDDIKTI Feeder application. The PDDIKTI Feeder application was developed based on a Web Service with a REpresentational State Transfer (REST) architecture. Web Service using REST Application Programming Interface (API), when a request to an endpoint is executed, it will get additional information that is not really needed. This is because when you access the endpoint, you will get all the data that was determined when the endpoint was developed. So that another filtering stage is needed to separate data that is not needed. The solution to overcome these problems in this research is trying to apply GraphQL. Test scenarios are created by setting up the syntax to access the PDDIKTI Feeder before using GraphQL or using only REST and comparing after implementing GraphQL. The experimental results show that the response time of GraphQL is 20% greater than that of the REpresentational State Transfer (REST). However, the file size response of GraphQL is only 10% compared to REST.**

Keywords—Feeder PDDIKTI, GraphpQL, REST, Web Service

I. INTRODUCTION

The Higher Education Database or Pangkalan Data Perguruan Tinggi (PDDIKTI) is a collection of data on the implementation of higher education that is nationally integrated. The data contained in the academic information system of a university must be entered in PDDIKTI [1]. The PDDIKTI Feeder application is one of the services that can be used by every university to help facilitate data input. The data input process through the PDDIKTI Feeder is easier and faster than manual input through the application. Integration of local information systems in universities can also be done with the PDDIKTI Feeder application so that system performance is more optimal [1]–[4]. However, the large and varied data on the implementation of higher education is an obstacle in the data input process into the PDDIKTI Feeder application [2]. The availability of a system that can support the interoperability of the PDDIKTI Feeder with the Higher Education information system is one solution to overcome this.

Several experiments related to access to the PDDIKTI Feeder application have been tried in previous research, including: integration of an integrated academic information system with the PDDIKTI Feeder [1] [2], development of single page applications on academic information systems [3], implementation of web services on student activity recording on PDDIKTI feeders [4]. Experiments in research [1] [2] [3] [4] only focus on implementing RESTful-based web services used by the PDDIKTI Feeder Application.

The PDDIKTI Feeder application was developed based on a Web Service with a REpresentational State Transfer (REST) architecture. Web Service using REST Application Programming Interface (API), when a request to an endpoint is executed, it will get additional information that is not really needed. This is because when you access the endpoint, you will get all the data that was determined when the endpoint was developed [5]. So that another filtering stage is needed to separate data that is not needed.

One solution to overcome these problems is by using the GraphQL approach. GraphQL is a new query language to implement a Web Service-based software architecture. The language is gaining momentum and is now used by large software companies, such as Facebook and GitHub [6][7][8]. In its implementation, GraphQL only requires one specific query that has determined its needs. The server will reply by providing data in Java Script Object Notation (JSON) format based on customized needs [5].

RESTful and GraphQL-based Web Service architecture has interesting characteristics to study, because several experiments showed varied results [9], [10]. Migration from REST to GraphQL has also been attempted in several studies [7], [11].

The purpose of this research is to model the GraphQL-based web service architecture on Higher Education Data services. Performance measurement of GraphQL and RESTful-based web service implementation is the main focus that will be studied in this research. The experiment was carried out by accessing one of the endpoints of the PDDIKTI Feeder using GraphQL and using RESTful. When conducting experiments, several parameters such as: response time [12] [13] [14] [15], CPU usage [14] [15], data size [14] [15] were measured to determine the performance of the two architectures.

II. RELATED WORK

Several experiments related to access to the PDDIKTI Feeder application have been tried in previous research, including: integration of an integrated academic information system with the PDDIKTI Feeder [1] [2], development of single page applications on academic information systems [3], implementation of web services on student activity recording

978-1-6654-6387-4/22/$31.00 ©2022 IEEE

on PDDIKTI feeders [4]. Experiments in research [1] [2] [3] [4] only focus on implementing RESTful-based web services used by the PDDIKTI Feeder Application.

The PDDIKTI Feeder application was developed based on a Web Service with a REpresentational State Transfer (REST) architecture. Web Service using REST Application Programming Interface (API), when a request to an endpoint is executed, it will get additional information that is not really needed. This is because when you access the endpoint, you will get all the data that was determined when the endpoint was developed [5]. So that another filtering stage is needed to separate data that is not needed.

One solution to overcome these problems is by using the GraphQL approach. GraphQL is a new query language to implement a Web Service-based software architecture. The language is gaining momentum and is now used by large software companies, such as Facebook and GitHub [6][7][8]. In its implementation, GraphQL only requires one specific query that has determined its needs. The server will reply by providing data in Java Script Object Notation (JSON) format based on customized needs [5].

RESTful and GraphQL-based Web Service architecture has interesting characteristics to study, because several experiments showed varied results [9], [10]. Migration from REST to GraphQL has also been attempted in several studies [7], [11].

III. System Design

There are 5 main stages carried out in this research, namely: system analysis, system architecture development, identification of hardware & software requirements, coding, implementation and measurement.

A. System Analysis

At this stage an analysis is carried out on the scope of the current system architecture related to NeoFeeder PDDIKTI and academic information systems contained in each university.

B. System Architecture Development

The implementation of GraphQL is the main focus that will be measured in the experiments in this study. GraphQL will be placed between the Academic Information System and NeoFeeder Client as shown in Figure 1.

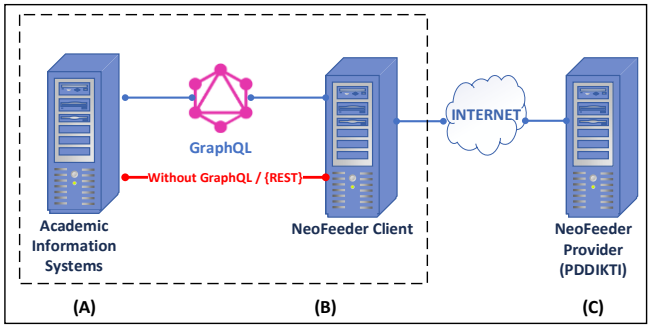

Fig. 1. Architecture System

C. Identification of Hardware and Software Requirements

At this stage, identification of the hardware and software needed for implementation refers to the system architecture that has been designed. The hardware specifications used are shown in Table I.

TABLE I. HARDWARE SPECIFICATIONS USED

No	Item	Description
1	CPU	CPU: Intel Core i5-6300U @ 4x 3GHz
2	Graphics	GPU: Mesa Intel (R) HD Graphics 520 (SKL GT2)
3	Memory	8 GB
4	Storage	128GB

Apart from hardware, some software is also needed in the experiments carried out in this study. The software specifications used in the experiment are shown in Table II.

TABLE II. SOFTWARE SPECIFICATIONS USED

No	Software	Version
1	Operating System	Manjaro 21.3.7 Ruah (Linux)
2	Kernel	x86_64 Linux 5.10.136-1-MANJARO
3	NodeJS	v14.18.2
4	NPM	6.14.15
5	ExpressJS	4.17.1
6	MongoDB	4.4.6
7	GraphQL	16.5.0
8	JMeter	5.5

D. Coding

At this stage, the program code (coding) for accessing the PDDIKTI Feeder is made by implementing graphQL compared to without using graphQL. Activities carried out at this stage:

1. Determine the endpoint to be accessed. **`GetListdosen`** is the endpoint that will be accessed in this experiment.
2. Experimental design

   The experiment was carried out by accessing the **`GetListdosen`** endpoint in two different ways, namely by applying graphQL and without applying graphQL. Endpoint access using graphQL is done by creating a query along with the specific fields to be accessed. The query is created in a graphQL compliant format. Endpoint access without using graphQL is done using a RESTful architecture. The experiment was carried out repeatedly with different number of requests: 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000.
3. Measurement

   Each endpoint call is recorded response times, the size of the response data in JSON format, CPU usage time. The experimental data are then inputted into tables and presented in graphical form.

E. Implementations and Measurements

At this stage, the software is installed on the prepared hardware. After the Server and Client are connected, then the database connection configuration is carried out on the client so that it is connected and the experimental process can be carried out.

IV. Experimental Result

Based on the system architecture shown in Figure 1, there are 3 main domains, namely: Academic Information System (A), NeoFeeder Client (B), NeoFeeder Provider PDDIKTI (C). In general, the main activities carried out at the implementation stage are as follows: first, access the PDDIKTI Neo Feeder Provider Web Service. This is done from NeoFeeder Client (B) to Neo Feeder Provider PDDIKTI (C) via the internet. The second stage, access the Web Service from the Academic Information System to the NeoFeeder

Client. This activity is the main focus carried out in research, consisting of two ways: by applying GraphQL and without using GraphQL.

A. Access PDDIKTI Neo Feeder Provider Web Service

The initial stage begins with the creation of program code to access the Neo Feeder Provider PDDIKTI web service. This is done to ensure each service can be accessed and used. The snippet of the API source code for accessing the PDDIKTI Neo Feeder Provider web service is shown in Figure 2.

```
import axios from 'axios';
import PddiktiToken from './models/pddikti-
token.js';
const buildConfig = (payload) => ({
  url: process.env.PDDIKTI_API_URL,  method:
'post',
  headers: { "Content-Type": "application/json",
},
  data: JSON.stringify(payload)
})
const getTokenRequest = async () => {
  const config = buildConfig({
    act: 'GetToken',
    username: process.env.PDDIKTI_USERNAME,
    password: process.env.PDDIKTI_PASSWORD,
  })
  try {
    const { data: { error_code, error_desc, data
} } = await axios(config);
    if (error_code === 0) {
      const pddiktiToken = new PddiktiToken({
value: data.token });
      await pddiktiToken.save();
    } else {
      throw new Error(error_desc + ` [error code
${error_code} : PDDIKTI]`);
    }
  } catch(err) {
    console.log(err);
  }
}
const pddiktiApi = async (payload) => {
  const existingPddiktiTokens = await
PddiktiToken.find().countDocuments();
  if (!existingPddiktiTokens) {
    await getTokenRequest();
  }
  const pddiktiToken = await
PddiktiToken.findOne().lean();
  const config = buildConfig({...payload, token:
pddiktiToken ? pddiktiToken.value : 'empty'});
  try {
    const { data } = await axios(config);
    const { error_code, error_desc } = data;
    if (error_code === 0) {
      return data;
    } else if (error_code === 100) {
      await
PddiktiToken.findByIdAndRemove(pddiktiToken._id)
;
      return pddiktiApi(payload);
    } else {
      throw new Error(error_desc + ` [error code
${error_code} PDDIKTI]`);
    }
  } catch (err) {
    console.log('Something went wrong:
pddiktiApi Function PDDIKTI');
  }
}
export default pddiktiApi;
```

Fig. 2. API Source Code Snippets for PDDIKTI Neo Feeder Provider Web Service Usage

Figure 2 shows a snippet of the API source code to access the Neo Feeder Provider PDDIKTI web service. Authentication of usernames, passwords, and getting tokens is done at this stage. The next step is to access web services from the Academic Information System to the NeoFeeder Client, which is the main focus of this research, in two ways: by applying GraphQL and without using GraphQL.

B. Access PDDIKTI Feeder without GraphQL

```
import pddiktiApi from './pddikti-api.js';
const app = express();
app.use('/pddikti/:action', async () => {
  const cpuUsageBefore = cpuUsage();
  const { action } = req.params;
  try {
    const { data: docs } = await pddiktiApi({ act:
action })

  const cpuUsageAfter = cpuUsage(cpuUsageBefore);
    res.status(200).send({
      message: 'Get pddikti data successfully',
      data: docs,
      cpuUsage: cpuUsageAfter,
    })
  } catch (err) {
    next(err);
  }
});
```

Fig. 3. Snippet of Neo Feeder API call source code without GraphQL

C. Access the PDDIKTI Feeder with GraphQL

```
import pddiktiApi from './pddikti-api.js';
const schema = buildSchema(`
  type Query {
    message: String
  }
  type CpuUsage {
    user: Int
    system: Int
  }
  type PDDIKTI {
    data: String
    cpuUsage: CpuUsage
  }
  type Mutation {
    pddikti(action: String): PDDIKTI
  }
`)
const root = {
  message: () => 'Hello World!',
  pddikti: async ({ action }) => {
    const cpuUsageBefore = cpuUsage();
    try {
     const { data } = await pddiktiApi({ act:
action });
      const cpuUsageAfter =
cpuUsage(cpuUsageBefore);
      return {
        data: JSON.stringify(data),
        cpuUsage: cpuUsageAfter
      };
    } catch(err) {
      console.log(err);
    }
  }
}

const app = express();
app.use('/graphql', graphqlHTTP({
  schema,
  rootValue: root,
  graphiql: true,
}));
```

Fig. 4. Snippet of Neo Feeder API call source code with GraphQL

Figures 3 and 4 show the source code snippets for accessing the Neo Feeder service, we can see there is a slight difference in access and programming. By using GraphQL we can insert queries so that the data response can be as desired. Meanwhile, without using GraphQL, there is direct access to the services available from Neo Feeder by using the end-point we want.

D. Data Response

1. Without GraphQL

```
URL: http://localhost:3000/pddikiti/GetListdosen

Result:
{
    "message": "Get pddikti data successfully",
    "data": [
        {
            "id_dosen": "dcb7791f-0346-4901-
8edc-014a6986ab7f",
            "nama_dosen": "PENGKI IRAWAN",
            "nidn": "[redacted]",
            "nip": "[redacted]",
            "jenis_kelamin": "L",
            "id_agama": 1,
            "nama_agama": "Islam",
            "tanggal_lahir": "[redacted]",
            "id_status_aktif": "1",
            "nama_status_aktif": "Aktif"
        },
        {
            "id_dosen": "2ecfed60-77be-48e7-
bc35-016dd3a6369b",
...
```

Fig. 5. Data Response without GraphQL

The results of the experiment in Figure 5 we can see that without the use of GraphQL, all data on the GetListDosen end-point including all its attributes will appear. This results in a lot of data that we don't need but the server still sends it. In addition to data, there are many other things that affect the speed of response and use of large resources.

2. With GraphQL

Unlike the case with the use of GraphQL in Figure 7, we can see that the use of queries is able to summarize the data that is really desired. In addition, the data sent, the speed of response and the use of resources are lighter. Another advantage of GraphQL is that if we need data from several end-points, then the request and response can only be done once in one action. It's different without the use of GraphQL, which means we have to make several requests at each desired end-point. An illustration of the comparison of the use of GraphQL can be seen in Figure 6.

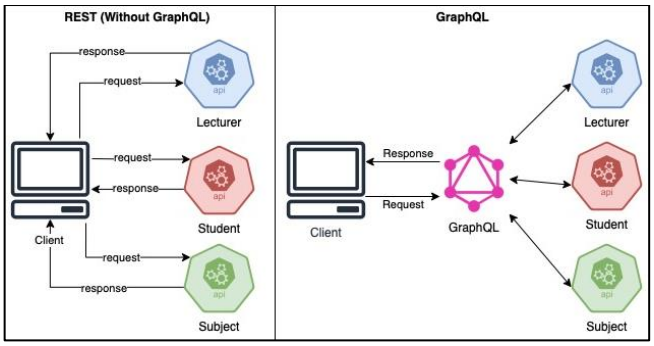

Fig. 6. Illustration of Comparison of Data Access Process Using and Not Using GraphQL

In the illustration in Figure 6 we can see that using REST without GraphQL means retrieving multiple resources requires multiple queries. In contrast to the use of GraphQL, it means that a single query restricts multiple resources.

```
URL: http://locahost:4000/graphql

Query:
mutation {
  pddikti(action: "GetListDosen") {
    id_agama
    id_dosen
    id_status_aktif
    jenis_kelamin
    nama_agama
    nama_dosen
    nama_status_aktif
    nidn
    nip
    tanggal_lahir
  }
}

Result:
{
  "data": {
    "pddikti": [
      {
        "id_agama": 1,
        "id_dosen": "dcb7791f-0346-4901-8edc-
014a6986ab7f",
        "id_status_aktif": "1",
        "jenis_kelamin": "L",
        "nama_agama": "Islam",
            "nama_dosen": "PENGKI IRAWAN",
            "nama_status_aktif": "Aktif",
            "nidn": "[redacted]",
            "nip": "[redacted]",
            "tanggal_lahir": "[redacted]"
        },
        {
            "id_agama": 1,
...
```

Fig. 7. Data Response with GraphQL

E. Query Response Times Measurement Results

Table III and Figure 8 present the results of measuring query response times in several times of testing the number of requests from 100, 200, to 1000 requests. The test results show that the response time using GraphQL is still less fast than using only REST at the end-point. This is because the use of GraphQL requires a process of adjusting the data to the desired one. When viewed from the response data obtained, of course GraphQL is superior because it fits the needs. In the 1000 request experiment there were anomalies that could be caused by unstable client computer conditions and this required special testing on different devices.

TABLE III. GRAPHQL RESPONSE TIME COMPARISON

Request Number	Without GraphQL(ms)	With GraphQL(ms)
100	292	231
200	298	234
300	303	258
400	314	293
500	377	374
600	618	850
700	996	1.575
800	1.298	4.651
900	5.985	11.000
1.000	18.202	11.862

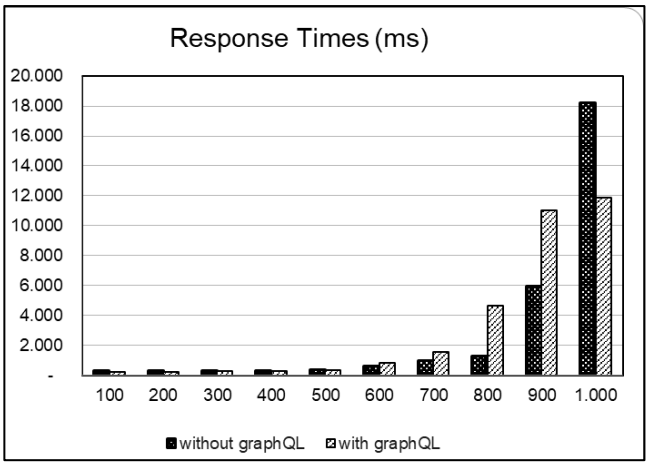

Fig. 8. GraphQL Response Time Comparison

F. Query Data Usage Measurement Results

TABLE IV. COMPARISON OF DATA USAGE

Request Number	Without graphQL(bytes)	With graphQL(bytes)
100	129.234	129.172
200	129.234	129.172
300	129.234	129.172
400	129.234	129.172
500	129.234	129.172
600	129.234	129.172
700	129.234	129.172
800	129.234	129.172
900	129.234	129.172
1.000	129.234	129.172

While in Table IV and Figure 9 GraphQL is superior because the data is not sent all, but in accordance with the query request from the client.

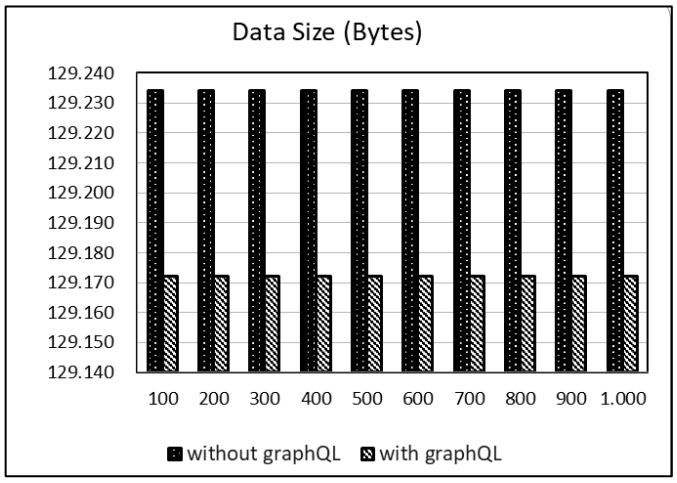

Fig. 9. Comparison of Data Usage

G. Query CPU Usage Measurement Results

TABLE V. CPU USAGE COMPARISON

Request Number	Without graphQL(seconds)	With graphQL(seconds)
100	26	40
200	25	42
300	61	46
400	20	27
500	22	29
600	33	32
700	20	22
800	30	21
900	38	24
1.000	25	23

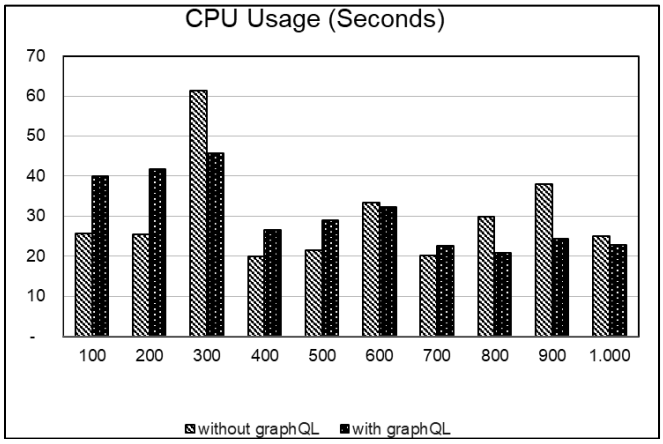

Fig. 10. CPU Usage Comparison

The last experiment is CPU usage, from several experiments in Table V and Figure 1o it can be seen that the use of GraphQL is still not stable. Several times the number of different requests presents different data, this needs to be studied more deeply about the factors that influence it. As from the influence of the internet and unstable networking.

V. CONSCLUSION

Based on the experimental results in research on the performance of using GraphQL on Neo Feeder services, it can be concluded that in terms of data usage, it is certain that GraphQL usage is superior and the data sent will be in accordance with the wishes of the user. In contrast, the response time and CPU Usage for GraphQL usage is still bad. This can be caused by several factors, including trials carried out directly through the internet network, the use of computer hardware must have adequate specifications.

To measure the performance of GraphQL, it is necessary to conduct more in-depth research and other scenarios that involve many end-points, so that in the future the advantages of using GraphQL in addition to Data Usage can be ascertained.

REFERENCES

[1] E. Widarti *et al.*, "INTEGRASI SISTEM INFORMASI AKADEMIK TERPADU (SIAKAT) DENGAN FEEDER PDDIKTI," vol. 2, no. 3, 2018.

[2] S. Widodo, H. Brawijaya, S. Samudi, and E. Retnoningsih,

"Integrasi Data Akademik Dengan Aplikasi Feeder PDDIKTI Berbasis Web service," *Bina Insa. ICT J.*, vol. 5, no. 2, pp. 153–162, 2018.

[3] P. L. L. Belluano, "Pengembangan Single Page Application Pada Sistem Informasi Akademik," *Ilk. J. Ilm.*, vol. 10, no. 1, pp. 38–43, 2018.

[4] V. H. Pranatawijaya, "Implementasi Pencatatan Aktivitas Mahasiswa Menggunakan Web Service Pada Feeder Pddikti Dengan Metode Extreme Programming," *J. Teknol. Inf. J. Keilmuan dan Apl. Bid. Tek. Inform.*, vol. 14, no. 2, pp. 179–188, 2020, doi: 10.47111/jti.v14i2.1188.

[5] K. I. E. Putra, "GrapgQL vs REST API: Apa bedanya?," 2019. [Online]. Available: https://www.dicoding.com/blog/graphql-api-vs-rest-api-apa-bedanya/. [Accessed: 25-Jul-2022].

[6] G. Brito and M. T. Valente, "REST vs GraphQL: A controlled experiment," *Proc. - IEEE 17th Int. Conf. Softw. Archit. ICSA 2020*, no. Dcc, pp. 81–91, 2020, doi: 10.1109/ICSA47634.2020.00016.

[7] G. Brito, T. Mombach, and M. T. Valente, "Migrating to GraphQL: A Practical Assessment," *SANER 2019 - Proc. 2019 IEEE 26th Int. Conf. Softw. Anal. Evol. Reengineering*, pp. 140–150, 2019, doi: 10.1109/SANER.2019.8667986.

[8] J. G. Ogboada, V. I. E. Anireh, and D. Matthias, "A Model for Optimizing the Runtime of GraphQL Queries," vol. 9, no. 3, pp. 11–39, 2021.

[9] D. A. Hartina, A. Lawi, B. Leonard, and E. Panggabean, "Performance Analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University," *2018 2nd East Indones. Conf. Comput. Inf. Technol.*, pp. 237–240, 2018.

[10] A. Lawi, B. L. E. Panggabean, and T. Yoshida, "Evaluating graphql and rest api services performance in a massive and intensive accessible information system," *Computers*, vol. 10, no. 11, 2021, doi: 10.3390/computers10110138.

[11] S. K. Mukhiya, F. Rabbiab, V. K. I. Punax, A. Rutle, and Y. Lamo, "A graphql approach to healthcare information exchange with hl7 fhir," *Procedia Comput. Sci.*, vol. 160, pp. 338–345, 2019, doi: 10.1016/j.procs.2019.11.082.

[12] R. Gunawan, A. Rahmatulloh, and I. Darmawan, "Performance evaluation of query response time in the document stored nosql database," in *2019 16th International Conference on Quality in Research, QIR 2019 - International Symposium on Electrical and Computer Engineering*, 2019, doi: 10.1109/QIR.2019.8898035.

[13] J. Sayago Heredia, E. Flores-García, and A. R. Solano, "Comparative Analysis Between Standards Oriented to Web Services: SOAP, REST and GRAPHQL," *Commun. Comput. Inf. Sci.*, vol. 1193 CCIS, pp. 286–300, 2020, doi: 10.1007/978-3-030-42517-3_22.

[14] M. Diego Casagranda França and E. Da Silva, "Performance Evaluation of REST and GraphQL APIs Searching Nested Objects," pp. 237–244, 2020, doi: 10.14210/cotb.v11n1.p237-244.

[15] E. Lee, K. Kwon, and J. Yun, "Performance Measurement of GraphQL API in Home ESS Data Server," *Int. Conf. ICT Converg.*, vol. 2020-October, pp. 1929–1931, 2020, doi: 10.1109/ICTC49870.2020.9289569.