TASK-RELEVANT API DEVELOPMENT FOR HIGHER EDUCATION USING GRAPHQL

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Dan Nygard

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Program:
Software Engineering

April 2024

Fargo, North Dakota

North Dakota State University
Graduate School

Title

TASK-RELEVANT API DEVELOPMENT FOR HIGHER EDUCATION USING GRAPHQL

By

Dan Nygard

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Anne Denton
Chair

Dr. Oksana Myronovych

Dr. Bernhardt Saini-Eidukat

Approved:

April 10, 2024	Dr. Simone Ludwig
Date	Department Chair

ABSTRACT

To develop applications that support a variety of campus needs, North Dakota State University's Enterprise Application Development team requires a method of accessing North Dakota State University System data related to university students, faculty, and staff. As state requirements limit direct access to this data, and conventional API access methods are not well-suited to application use cases, this paper will explore how the data is acquired, stored, and then made accessible to individual applications using GraphQL. A single application, Graduate Waiver Wire, is presented as a use case depicting how GraphQL aids in the automatic data update process, freeing the time previously spent by Graduate School personnel in manually updating graduate student information.

ACKNOWLEDGMENTS

I would not be writing this if not for my advisor, Dr. Anne Denton. Her guidance, support, and encouragement throughout all aspects of this project have been invaluable. I thank her for challenging me, from problem statement to finished project. It has also been an honor to have Dr. Oksana Myronovych and Dr. Bernhardt Saini-Eidukat serve on my committee. To all of them, I appreciate the time spent and support provided during this project. And I also appreciate the mentorship, wisdom, and insight shared throughout many conversations that had nothing to do with my Master's paper.

I would also like to thank my teammates at NDSU Enterprise Application Development, William Diederich, Steve Templeton, and Trevor Knutson. It is a joy to work with each of them. Richard Frovarp, Principal Software Engineer and Team Lead, has made us all better software engineers by sharing his expertise, and not sharing it when he knows we can figure it out. And I would like to thank our leadership, namely Steve Sobiech and Marc Wallman, for their steadfast support and confidence in us.

Thanks to Nonnie Tangen, Alexandra Wede, Samantha Akers, and Audra Hart at the NDSU Graduate School for working with me to develop Graduate Waiver Wire. The software is a successful, working product because of their input.

And my most important thanks go to my family: my parents, Gordon and Theresa Nygard, my brother Leif and sister Amanda, my in-laws Pam and Steve Reis, and brother-and-sister-in-law Cody and Brittany Leadbetter. A special thanks to Dr. Kendall Nygard, who provided so much encouragement to start the Master's program.

To Nicole, you are an incredible wife and partner, and the love of my life. Here's hoping things slow down just a little.

To Genevieve and Gerald, I couldn't ask for better children. You amaze me each day with your creativity and curiosity. I hope I have set a good example, and as you grow up and move through life you keep finding your own new mountains to climb.

TABLE OF CONTENTS

ABSTRACT....................iii

ACKNOWLEDGMENTS....................iv

LIST OF TABLES....................viii

LIST OF FIGURES....................ix

1. INTRODUCTION....................1

  1.1. Problem Statement....................1

2. BACKGROUND....................3

  2.1. Application Development and Data Access at NDSU....................3

    2.1.1. Application Development....................3

    2.1.2. Data Access Overview....................3

  2.2. GraphQL Overview....................4

    2.2.1. Specification....................4

    2.2.2. Query....................5

    2.2.3. Schema Development....................7

  2.3. Related Work....................7

3. IMPLEMENTATION: BIODEMO GRAPHQL SERVER....................9

  3.1. Base Server....................9

  3.2. Biodemo GraphQL Server....................9

    3.2.1. Schema Definition....................9

    3.2.2. Query Definition....................10

    3.2.3. Class Definition....................11

    3.2.4. Resolver Implementation....................13

    3.2.5. The N+1 Problem....................15

  3.3. Use Case: Graduate Waiver Wire....................16

3.3.1. Background....16

3.3.2. Graduate Waiver Wire Specification....17

3.3.3. Data Structure....17

3.3.4. GraphQL Query....18

3.3.5. POJO Classes....20

3.3.6. Biodemo Service Implementation....23

4. DISCUSSION AND CONCLUSION....27

4.1. Discussion....27

4.1.1. Update Time....27

4.1.2. Benefits Related to GraphQL....28

4.1.3. Drawbacks of GraphQL....29

4.2. Conclusion....29

REFERENCES....30

LIST OF TABLES

Table Page

4.1. GraphQL Response and Refresh Database Times in Seconds..28

LIST OF FIGURES

Figure Page

2.1. Data flow from NDUS Peoplesoft to an NDSU EAD application....................................3

2.2. Example GraphQL query [4]..5

2.3. Query calling for a data object along with related objects [4]...6

2.4. Query with an argument [4]..6

3.1. A HrData and StudentPlan GraphQL schema definition...10

3.2. User class diagram...12

3.3. A HrData.groovy class definition...12

3.4. Calling a list of HrData objects in UserResolver.groovy..13

3.5. Calling a list of User objects filtered by job family in UserQueryResolver.groovy..........14

3.6. Queries demonstrating nesting of data..16

3.7. Graduate Waiver Wire data diagram for students, programs, and positions. Data automatically updated using a GraphQL query is in bold..18

3.8 Graduate Waiver Wire GraphQL query for users (students), plans (programs), and positions (hrData)...19

3.9. Example student response object...20

3.10. Plain Old Java Objects for JSON deserialization (top), and their corresponding Apache Cayenne data model classes (bottom). A GradUser has one-to-many relationships with HrData and StudentPlan objects; the Student class correspondingly has one-to-many relationships with the Position and Program class..22

3.11. Deserializing JSON into a list of GradUser objects...22

3.12. Method to update a Student based on a GradUser object...23

3.13. BiodemoServiceImpl class...24

1. INTRODUCTION

North Dakota State University’s Enterprise Application Development (EAD) team serves a variety of campus community needs, including application development for individual departments and programs. In addition to maintenance on production applications, EAD must also be prepared to develop new applications as campus needs arise. These applications rely on accessing data relevant to the campus community. Automating this flow of data requires retrieving information from its original source, North Dakota University System (NDUS) PeopleSoft records, storing it, and providing a way for applications to easily access it.

This paper will focus on the last step mentioned above: providing a method for EAD applications to easily access stored data. In doing so, it will discuss how NDSU has adopted GraphQL in order to make complex data accessible to applications within a single API call.

1.1. Problem Statement

The objective of EAD is to create a task-relevant set of information sourced from the NDUS PeopleSoft API. The North Dakota University System only allows for PeopleSoft data to be accessed within two two-hour time intervals each day, precluding direct access to this data. Individual applications need to access diverse subsets of this data to fit their particular needs. Thus, there is a need for a local API service and data storage method that will allow applications to make straightforward data requests that return only what is needed by that application. In addition, as university applications require information pulled from multiple tables (for example, a call for student information may also require the student's academic programs), the service should have the ability to make complex requests without accessing numerous API endpoints. Finally, to improve application maintenance, the service should provide a consistent API client language that all developers on the team can understand and maintain across numerous projects.

Due to the above, conventional API calling methods are not well-suited. REST services are prone to overfetching (returning more data than what is needed) and require multiple calls to return data if it exists in more than one table [1]. And while SQL can handle complex queries, EAD would prefer to provide applications a limited set of data through an abstraction layer rather than a direct connection to the database. To provide this abstraction layer and a straightforward query language to access it, EAD has chosen to use GraphQL, and this paper will demonstrate how GraphQL helps fulfill the diverse data needs of a university application development department.

2. BACKGROUND

2.1. Application Development and Data Access at NDSU

2.1.1. Application Development

EAD currently runs 23 custom web applications. These applications serve a variety of university needs, including research equipment inventory tracking, signing up for training events, and identity and access management support. Every application accesses a different data set, and future applications will have their own data requirements, specialized to fit that particular application's needs.

2.1.2. Data Access Overview

The original source of NDSU student data is the NDUS PeopleSoft Campus Solutions system, and the source of faculty and staff data is PeopleSoft Human Capital Management. As PeopleSoft data can only be accessed within certain time windows, they cannot be directly accessed by applications in an efficient manner. EAD therefore maintains a MariaDB database to store this information, and a series of data feed programs to access PeopleSoft within the prescribed time windows and populate the database with current student, staff, and faculty data. An overview of the flow of data from NDUS PeopleSoft to an EAD application is provided in Figure 2.1.

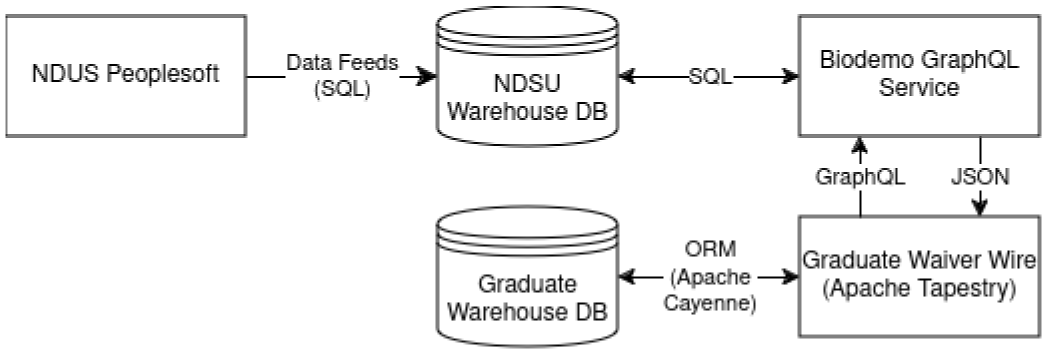


Figure 2.1. Data flow from NDUS PeopleSoft to an NDSU EAD application.

Figure 2.1 depicts how NDUS PeopleSoft data is fetched by the data feeds, which are written in the Groovy programming language and use the groovy-sql module [2] to write the data to the NDSU Warehouse database. Here, the need arises to make this data available in a flexible, efficient manner to NDSU applications, and EAD has chosen GraphQL for this purpose. When an application such as Graduate Waiver Wire uses the GraphQL query language to make a data request, the Biodemo GraphQL service uses groovy-sql to interact with NDSU Warehouse, and will return the requested data to Graduate Waiver Wire in the form of a JSON string. Graduate Waiver Wire then processes the returned data and stores it in the Graduate Warehouse database using Apache Cayenne Object-Relational Mapping [3].

The main contribution of this paper is in the interactions between the Biodemo GraphQL service and the Graduate Warehouse database. The structure of this interaction is discussed in detail in Chapter 3.

2.2. GraphQL Overview

GraphQL is an API access method that is best described in two parts. On the client side, it is a query language built to access data from an API. On the server side, it is a schema for defining data along with a runtime engine which responds to client queries [4]. Its underlying design and philosophy is outlined in a specification document [5] which will be discussed in the next section.

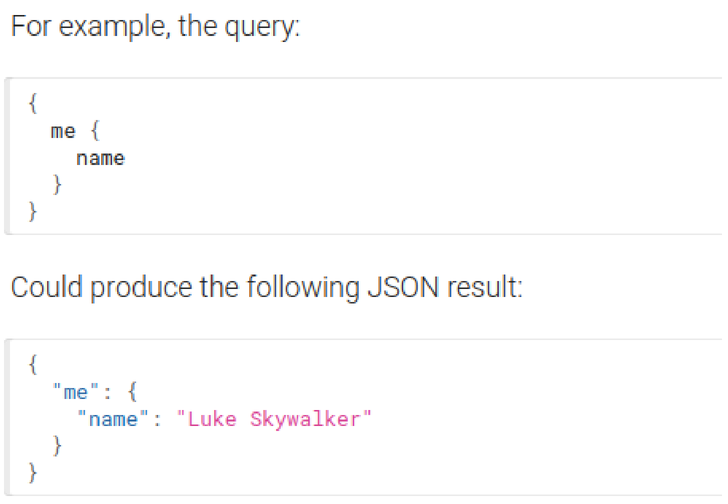
2.2.1. Specification

The initial GraphQL specification was developed by Facebook in 2012 and made open source in 2015. Notably, it defines GraphQL as "not a programming language capable of arbitrary computation, but ... a language used to make requests to application services that have capabilities defined in this specification." In other words, the specification provides the rules and principles for building services useful to GraphQL clients. No programming language or data storage system is mandated [5] and GraphQL libraries and frameworks have been developed for languages such as Java, Javascript, Python, C#, and Ruby [6].

The GraphQL specification outlines three types of operations that may be conducted by a client: query, mutation, and subscription. A query is a read-only call for data from the server; a mutation exists to write data to the server and can also fetch data in the same call; and a subscription is "a long-lived request that fetches data in response to source events." [5] As the primary EAD use cases involve requesting data, the next section will focus on the GraphQL Query operation.

2.2.2. Query

A GraphQL query is a String object which outlines the information being requested by a client. The server will receive the query, check to ensure it is requesting only items that have been outlined in the schema, and return the requested data once the check clears, as shown in Figure 2.2 from the GraphQL Foundation tutorial [4].


For example, the query:

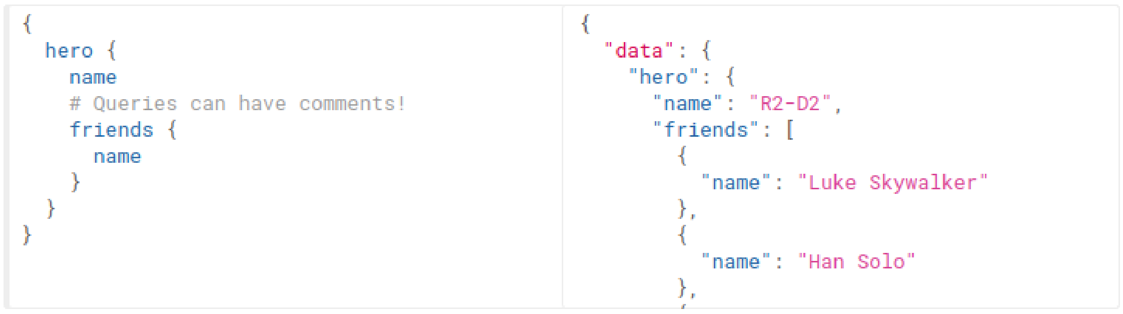
```
{
  me {
    name
  }
}
```

Could produce the following JSON result:

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

Figure 2.2. Example GraphQL query [4].

Queries can pull sub-objects related to the main object being requested, as shown in Figure 2.3. This allows for a straightforward method to present complex data objects.



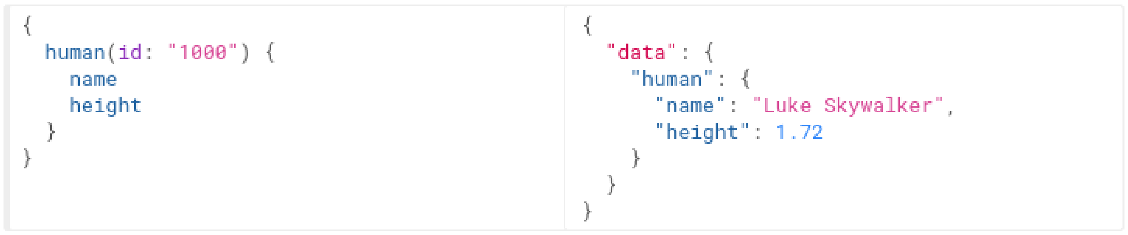
```
{
  hero {
    name
    # Queries can have comments!
    friends {
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
```

Figure 2.3. Query calling for a data object along with related objects [4].

In Figure 2.3 the hero object has its own field (name) but it also has a friends field, in which multiple other heroes are included. If another class is defined, for example, Spaceships, then it can be set up so each Hero object can have one or many Spaceships. This then allows for later deserialization of query responses into classes which incorporate composition. In object-oriented programming, composition is a concept in which one class shows it "has" members of another class by declaring the owned class as one of its variables. These are compositional relationships, in which one object can contain another object, and a GraphQL-based API is useful in calling data structured in this manner.

Queries may also pass arguments directly. For example, a query can be developed in which an ID parameter is passed in, and only objects assigned to that ID will be returned [4], as shown in Figure 2.4.



```
{
  human(id: "1000") {
    name
    height
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 1.72
    }
  }
}
```

Figure 2.4. Query with an argument [4].

2.2.3. Schema Development

A schema in GraphQL is created on the server side, and provides an explanation of the overall data set that will be accessed. In it, an object is described in terms of its fields, each of which is given a type. Types can be, among other things, scalar values (Int, Float, String, Boolean, and ID, along with custom values that can be defined by the programmer). In addition, another defined object in the schema can be used as a type definition. For example, in the below basic schema two objects are defined:

```
type Student {
        id: ID!
        name: String
        classes: [Class]
}

type Class {
        id: ID!
        title: String
}
```

In this schema, a student is able to hold multiple Class objects, which is shown by placing the return value in square brackets. One-to-one relationships can also be given in a schema: if the student were limited to only one class, this could be shown by removing the square brackets.

2.3. Related Work

GraphQL was built by Facebook (now Meta) as a response to difficulties in implementing a REST architecture while developing the Facebook mobile application [7]. It is now open source and governed by the GraphQL Foundation [8], and used by a large number of companies, including AirBnB, Atlassian, GitHub, Meta, and PayPal [9]. Netflix has also implemented GraphQL and developed the Domain Graph Services (DGS) Framework as an implementation of the GraphQL specification [10].

GraphQL has also found uses in archival metadata systems. For example, it is used by the European Holocaust Research Infrastructure portal, which makes available archival material from 500 different research institutions [11], and by researchers working with healthcare metadata [12] and developing federated research query services [13]. Work has also been done to automate schema development [14] and pair GraphQL with the Linked Open Data Cloud [15].

Work has been done to provide performance and feature comparisons between GraphQL and the REST framework. These experiments produce a variety of conclusions. For example, Hartina and Panggabean analyze GraphQL and REST web protocols built to access similar data from the Institute for Research and Community Service at Hansanuddin University, noting that while REST produced faster response times, GraphQL reduces over-fetching of data and is the right choice in situations where data needs change often [16]. Brito and Valente study computer science student ability to implement queries in both protocols, showing that GraphQL requires less developer effort to implement [17]. In a qualitative survey, Vadlamani et. al. surveyed developers and noted that developers liked GraphQL's strong typing, validation, and schema testing tools as opposed to the lack of these for REST; however, some developers in the study did not like GraphQL's learning curve and potential computing complexity (for example, in terms of N+1 problems) [18]. The N+1 problem in relation to GraphQL applications is discussed further in Section 3.2.5.

As a final note, Hartig and Pérez have conducted an analysis of the properties of GraphQL's query language to develop formal mathematical definitions of properties (such as a schema and graph), and to "define a logical data model that formally captures the notion of this graph" which allows further study of "the computational complexity of the language" [19], [20].

3. IMPLEMENTATION: BIODEMO GRAPHQL SERVER

3.1. Base Server

The base server implementation, which the EAD ‘Biodemo’ server implements, is a Spring Boot application built using the Netflix Domain Graph Service (DGS) framework [21] which incorporates the GraphQL Java implementation [22] and has been converted to the Java-based Groovy programming language.

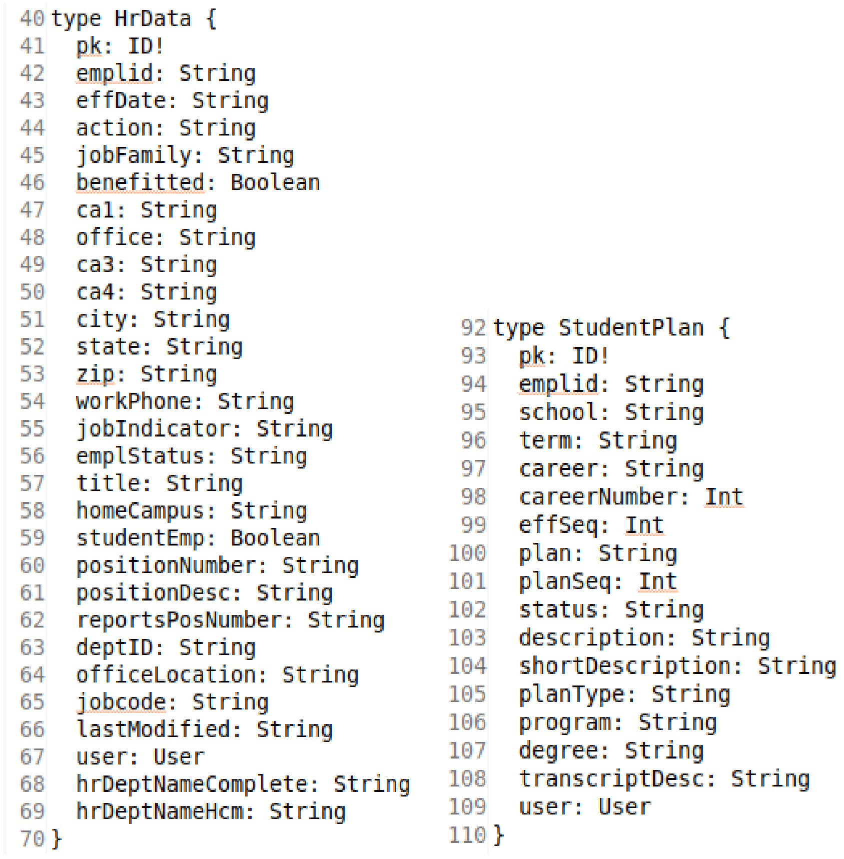
3.2. Biodemo GraphQL Server

As given in its name, the Biodemo GraphQL server is intended to provide biographical and demographic information about NDSU people.

3.2.1. Schema Definition

The data to be fetched is delineated as a GraphQL schema in the user.graphqls file. The base object in the schema is the User, and a User can be a student, staff, or faculty member. In the schema, the fields of a User object are described, and given their data types. The emplid (employee ID) field serves as the primary key of the object, and other information related to the User is included such as first, middle, and last name. While standard scalar types such as Strings are in the majority, types have also been created to represent other defined objects related to a User's campus experience such as employment positions; student programs, plans, and subplans of study; and information regarding a student's terms of enrollment.

To observe how the data is interconnected, Figure 3.1 shows two of the related objects a User may hold:



```
40 type HrData {
41   pk: ID!
42   emplid: String
43   effDate: String
44   action: String
45   jobFamily: String
46   benefitted: Boolean
47   ca1: String
48   office: String
49   ca3: String
50   ca4: String
51   city: String
52   state: String
53   zip: String
54   workPhone: String
55   jobIndicator: String
56   emplStatus: String
57   title: String
58   homeCampus: String
59   studentEmp: Boolean
60   positionNumber: String
61   positionDesc: String
62   reportsPosNumber: String
63   deptID: String
64   officeLocation: String
65   jobcode: String
66   lastModified: String
67   user: User
68   hrDeptNameComplete: String
69   hrDeptNameHcm: String
70 }
```

```
 92 type StudentPlan {
 93   pk: ID!
 94   emplid: String
 95   school: String
 96   term: String
 97   career: String
 98   careerNumber: Int
 99   effSeq: Int
100   plan: String
101   planSeq: Int
102   status: String
103   description: String
104   shortDescription: String
105   planType: String
106   program: String
107   degree: String
108   transcriptDesc: String
109   user: User
110 }
```

Figure 3.1. A HrData and StudentPlan GraphQL schema definition.

Here, a User has bidirectional relationships with HrData and StudentPlan, as each HrData and StudentPlan object is given a particular User in lines 67 and 109. This provides flexibility, in that a User query can provide information on that user's HrData and StudentPlan objects, while a HrData or StudentPlan query can provide information on the User assigned to each particular object returned.

3.2.2. Query Definition

Queries are defined in a schema to provide a basic framework of what calls the server will be able to answer. The majority of queries currently defined relate to an NDSU person (called a User in the schema), and can call a single User based on their `emplid` or `username` fields. Queries are also defined to return multiple users. For example, the query:

```
usersByJobFamilies(families: [String]): [User]
```

will return all users who have positions within a list of job family parameters. This query will be employed in Section 3.3 to select all people holding positions in the job families related to graduate assistantships.

3.2.3. Class Definition

Groovy classes are then implemented to reflect the scalar types each object holds. For example, the User.groovy class defines its fields to match by name and type the scalar fields defined in the schema, and provides an empty constructor along with a constructor that takes in a `row` parameter, which is a single row in a database "SELECT" call. Only scalar types are defined here, as these classes exist to process the results of the database calls that will be made by the resolver classes. As the User schema object was defined by multiple EAD team members, it is depicted as a class diagram in Figure 3.2.

In Figure 3.3, The HrData schema and its scalar fields are represented by the HrData.groovy class, which is given in its entirety.

Each individual class handles each scalar field type for every object defined in the schema. If a field has types defined as other objects (for example, the User schema has HrData, and vice versa), the User class will rely on the HrData class definition in a call that will be implemented in a Resolver, which will be discussed in the next section.

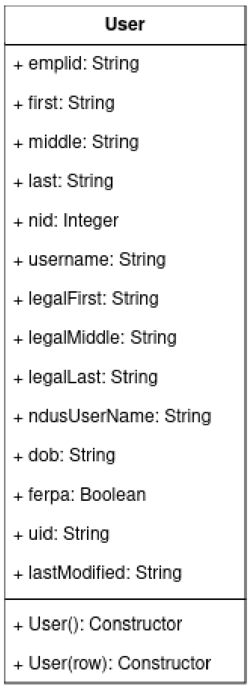
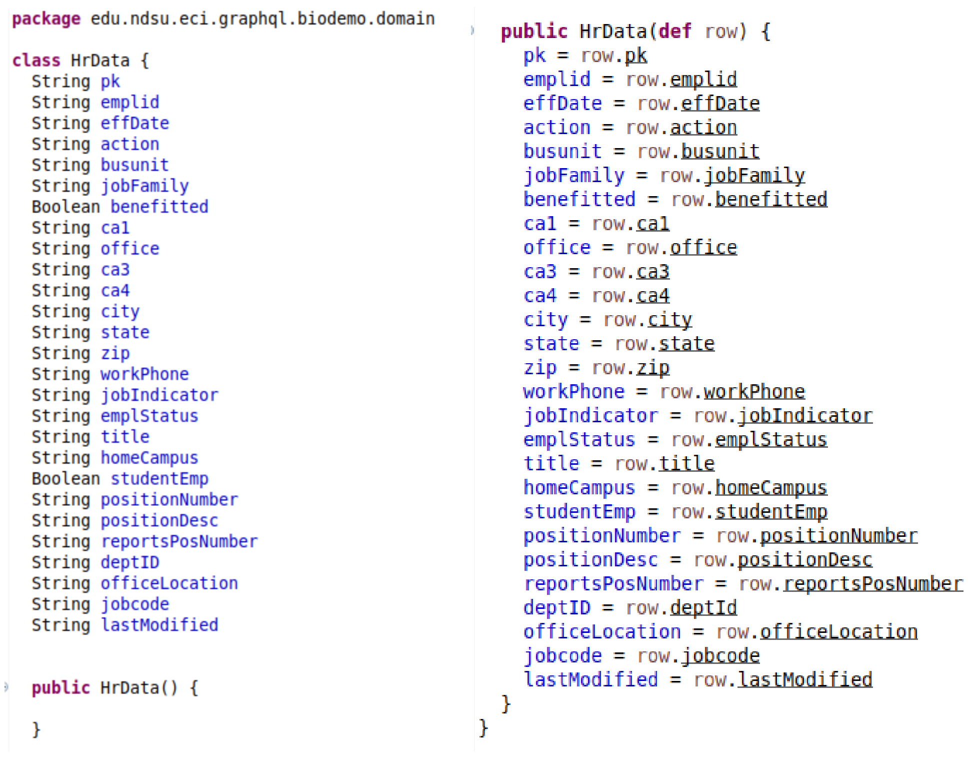


Figure 3.2. User class diagram.

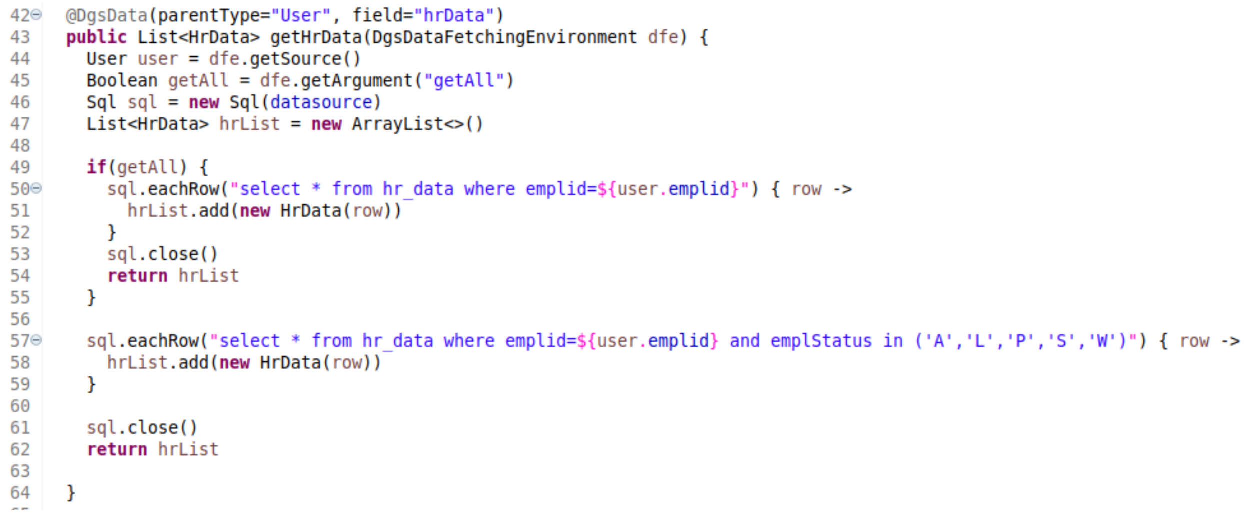


```
package edu.ndsu.eci.graphql.biodemo.domain

class HrData {
  String pk
  String emplid
  String effDate
  String action
  String busunit
  String jobFamily
  Boolean benefitted
  String ca1
  String office
  String ca3
  String ca4
  String city
  String state
  String zip
  String workPhone
  String jobIndicator
  String emplStatus
  String title
  String homeCampus
  Boolean studentEmp
  String positionNumber
  String positionDesc
  String reportsPosNumber
  String deptID
  String officeLocation
  String jobcode
  String lastModified


  public HrData() {

  }

  public HrData(def row) {
    pk = row.pk
    emplid = row.emplid
    effDate = row.effDate
    action = row.action
    busunit = row.busunit
    jobFamily = row.jobFamily
    benefitted = row.benefitted
    ca1 = row.ca1
    office = row.office
    ca3 = row.ca3
    ca4 = row.ca4
    city = row.city
    state = row.state
    zip = row.zip
    workPhone = row.workPhone
    jobIndicator = row.jobIndicator
    emplStatus = row.emplStatus
    title = row.title
    homeCampus = row.homeCampus
    studentEmp = row.studentEmp
    positionNumber = row.positionNumber
    positionDesc = row.positionDesc
    reportsPosNumber = row.reportsPosNumber
    deptID = row.deptId
    officeLocation = row.officeLocation
    jobcode = row.jobcode
    lastModified = row.lastModified
  }
}
```

Figure 3.3. A HrData.groovy class definition.

3.2.4. Resolver Implementation

Where the Object classes exist to hold schema data, Resolver classes make the actual calls to the database, reflecting the schema query definitions. The User resolver is separated into two parts, defined by the UserResolver.groovy and UserQueryResolver.groovy classes. In UserResolver.groovy, methods are provided to handle object field types defined in the User schema. This is done because separate database calls are required for each object. For example, in Figure 3.4 the HrData information is fetched:



```
42  @DgsData(parentType="User", field="hrData")
43  public List<HrData> getHrData(DgsDataFetchingEnvironment dfe) {
44    User user = dfe.getSource()
45    Boolean getAll = dfe.getArgument("getAll")
46    Sql sql = new Sql(datasource)
47    List<HrData> hrList = new ArrayList<>()
48
49    if(getAll) {
50      sql.eachRow("select * from hr_data where emplid=${user.emplid}") { row ->
51        hrList.add(new HrData(row))
52      }
53      sql.close()
54      return hrList
55    }
56
57    sql.eachRow("select * from hr_data where emplid=${user.emplid} and emplStatus in ('A','L','P','S','W')") { row ->
58      hrList.add(new HrData(row))
59    }
60
61    sql.close()
62    return hrList
63
64  }
```

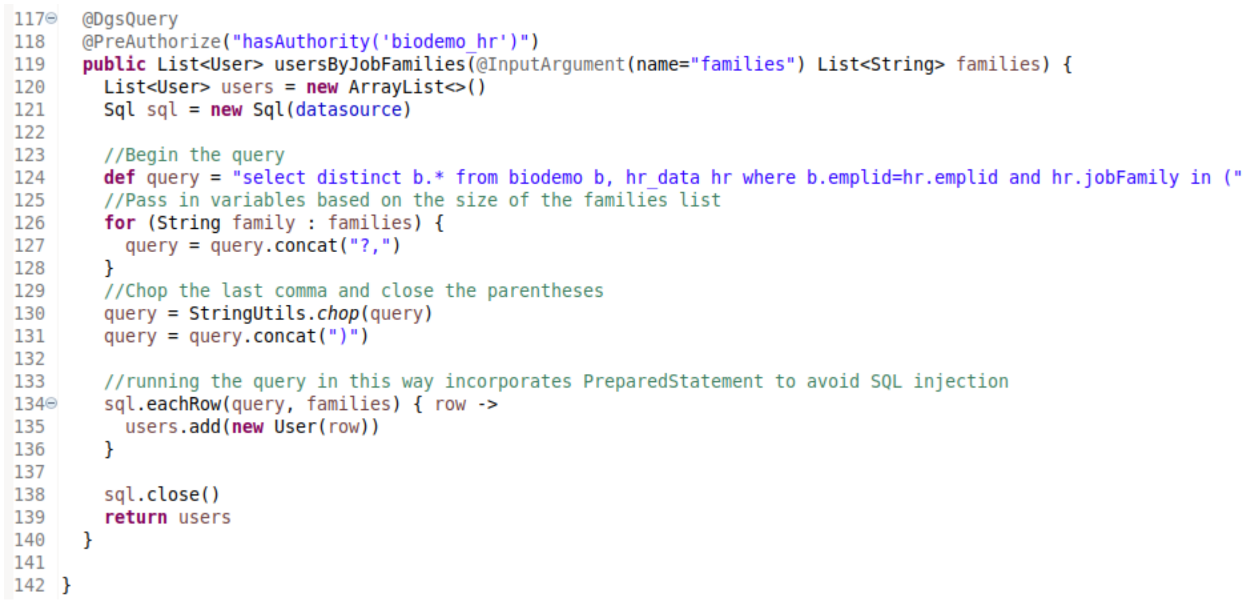
Figure 3.4. Calling a list of HrData objects in UserResolver.groovy.

Here, the Netflix `DGS @DgsData` annotation defines that this query parent is User who will be looking for HrData objects. The `DgsDataFetchingEnvironment` parameter allows for greater flexibility in defining queries: here, it allows for an optional parameter `getAll` to define what data will be returned (line 45). If `getAll` is true in the defined query, then the SQL query will return all data. If false, it returns data only within certain `emplStatus` categories, observable in the SQL call in line 57.

Similar fetching methods are then given for each User field type that is an object (as defined in the original schema): `getWorkStudy`, `getStudentPrograms`, `getStudentPlans`, `getStudentSubplans`, and `getStudentCarTerm`. The DGS GraphQL framework will look for these method implementations on startup, and will throw exceptions and halt program execution if they are not there.

With each schema field relating to an object given a resolution in UserResolver.groovy, the UserQueryResolver.groovy class is then prepared to execute the User queries defined in the schema. All scalar types for the User are handled through the User.groovy class, while the object types are handled by the resolutions built in UserResolver.groovy.

As an example, a list of Users filtered by job family is implemented by passing in a list of job family strings as a "families" parameter, as shown in line 119 of Figure 3.5:



```
117   @DgsQuery
118   @PreAuthorize("hasAuthority('biodemo_hr')")
119   public List<User> usersByJobFamilies(@InputArgument(name="families") List<String> families) {
120     List<User> users = new ArrayList<>()
121     Sql sql = new Sql(datasource)
122 
123     //Begin the query
124     def query = "select distinct b.* from biodemo b, hr_data hr where b.emplid=hr.emplid and hr.jobFamily in ("
125     //Pass in variables based on the size of the families list
126     for (String family : families) {
127       query = query.concat("?,")
128     }
129     //Chop the last comma and close the parentheses
130     query = StringUtils.chop(query)
131     query = query.concat(")")
132 
133     //running the query in this way incorporates PreparedStatement to avoid SQL injection
134     sql.eachRow(query, families) { row ->
135       users.add(new User(row))
136     }
137 
138     sql.close()
139     return users
140   }
141 
142 }
```

Figure 3.5. Calling a list of User objects filtered by job family in UserQueryResolver.groovy.

Within the resolver, the below SQL statement is built in lines 124-131 and executed, in this case, for a list of job families containing five elements. The question marks indicate String placeholders, and `families` is the list of job families to fill each placeholder:

```
("select distinct b.* from biodemo b, hr_data hr
where b.emplid=hr.emplid and hr.jobFamily in (?,?,?,?,?)", families)
```

This returns all users who have positions within the five job families passed in. The above demonstrates the granularity that can be built into a schema-defined call. This call chooses a person from the `biodemo` table, but also relies on related `jobFamily` information tied to that student from the `hr_data` table. This returns a list of students filtered by `jobFamily`, thus selecting only students in the assistantship job families and filtering out students who may be working in other capacities. This query will be used in Section 3.3, which demonstrates GraphQL employed in an EAD application.

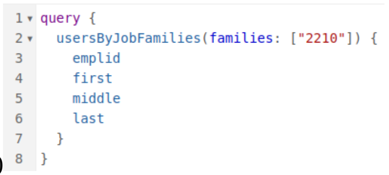
3.2.5. The N+1 Problem

Mentioned briefly in Related Work, the N+1 problem can occur when working with nested data in GraphQL. In essence, if data in a GraphQL schema exists in two separate tables, multiple database calls must be made (the original N, plus one for each subsequent set of nested data for each object returned in N). As noted in Roksela et. al:

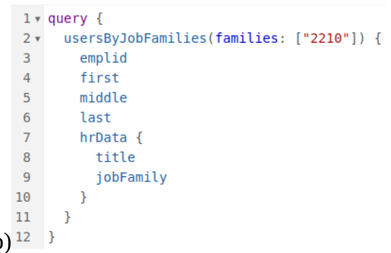
> A typical scenario looks as follows: the first database query returns a list of references (1 query returns N objects), and then N queries are made to access one of the objects each, causing in total N + 1 database queries. [23]

While GraphQL is built to respond to complex queries, ignoring potential N+1 problems can cause significant performance decline. As an example, the queries in Figure 3.6 demonstrate three separate calls, each requesting an additional layer of nested data.

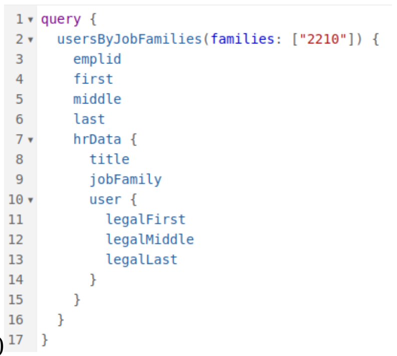
In Figure 3.6 (a) a query of Users only asks for User information. In (b), HR information is also requested. While requesting data structured in this manner is a useful feature, it does necessitate another database call for each separate user. To illustrate a potential problem, in (c) a User is requesting HR information, but within the HR information further User information is requested. Thus, in (c) for each individual User, each individual HR object that User holds will also trigger a database call. Nesting queries in this manner can greatly increase response times.

(a)


```
query {
  usersByJobFamilies(families: ["2210"]) {
    emplid
    first
    middle
    last
  }
}
```

(b)


```
query {
  usersByJobFamilies(families: ["2210"]) {
    emplid
    first
    middle
    last
    hrData {
      title
      jobFamily
    }
  }
}
```

(c)


```
query {
  usersByJobFamilies(families: ["2210"]) {
    emplid
    first
    middle
    last
    hrData {
      title
      jobFamily
      user {
        legalFirst
        legalMiddle
        legalLast
      }
    }
  }
}
```

Figure 3.6. Queries demonstrating nesting of data.

In addition to query design, numerous other N+1 mitigation strategies exist. For example, Robinson discusses minimizing N+1 issues through schema design, resolver optimization, and caching [24].

In the use case described in Section 3.3, the data set is compact enough that N+1 doesn't impact performance in a way that requires further mitigation, and our current method is straightforward and easy for developers to understand, which improves maintainability. However, N+1 is an important potential roadblock to be aware of when implementing GraphQL systems.

3.3. Use Case: Graduate Waiver Wire

3.3.1. Background

The NDSU College of Graduate and Interdisciplinary Studies (or, the Graduate School) requested that EAD develop an application to track graduate student tuition waivers. This application, Graduate Waiver Wire, had a requirement to automatically include a record for each NDSU graduate student holding a tuition waiver. This automatic student record would have to contain student name and emplid; information from HR regarding each student's assistantships (including department name, employment status, and so forth); and the description (in other words, the working name) and term of each graduate program in which that particular student is enrolled. All other information (related to contracts, the waivers themselves, additional hours worked, and so forth) for each student would be manually entered by Graduate School staff into fields existing alongside the automatically-created record.

GraphQL was chosen for this activity because of its flexibility and ease of handling complex data. For example, consider a student who has two different assistantships and three different programs (for example, two master's degrees and a certificate). Where REST requires multiple individual requests to fetch this data, and will include all fields for each of the three tables requested, GraphQL returns a single JSON object containing the student, their programs, and their positions, and only the fields within those three subjects that are needed by the application. Once received, this data can then be parsed into java objects on the client side.

3.3.2. Graduate Waiver Wire Specification

Graduate Waiver Wire is built using the Model View Controller architecture within the Apache Tapestry web framework, which uses Java as its primary programming language. Object relational mapping to the database is handled by Apache Cayenne, which defines a Java class within Graduate Waiver Wire for each data table.

Google's Gson is an open-source library for serializing and de-serializing Java code objects to JSON [25]. It is used by Graduate Waiver Wire to handle GraphQL response strings.

3.3.3. Data Structure

A MariaDB database called graduate_warehouse has been created for Graduate Waiver Wire data persistence. While graduate_warehouse contains other tables related to graduate student tuition waivers, for purposes of this case study the following sections will focus solely on the tables containing data requested from the GraphQL server.

Figure 3.7 illustrates the three data tables that the automated system will be updating: students, programs, and positions. Each table in Figure 3.7 contains data that is automated through GraphQL, along with data that is manually entered (or kept up-to-date within the system). For example, student emplid, first, middle, and last names are automated; however, the "eligibleGrader" and "eligibleTa" attributes are determined by Graduate School personnel and manually controlled. For each table, the data that is automated is in bold.

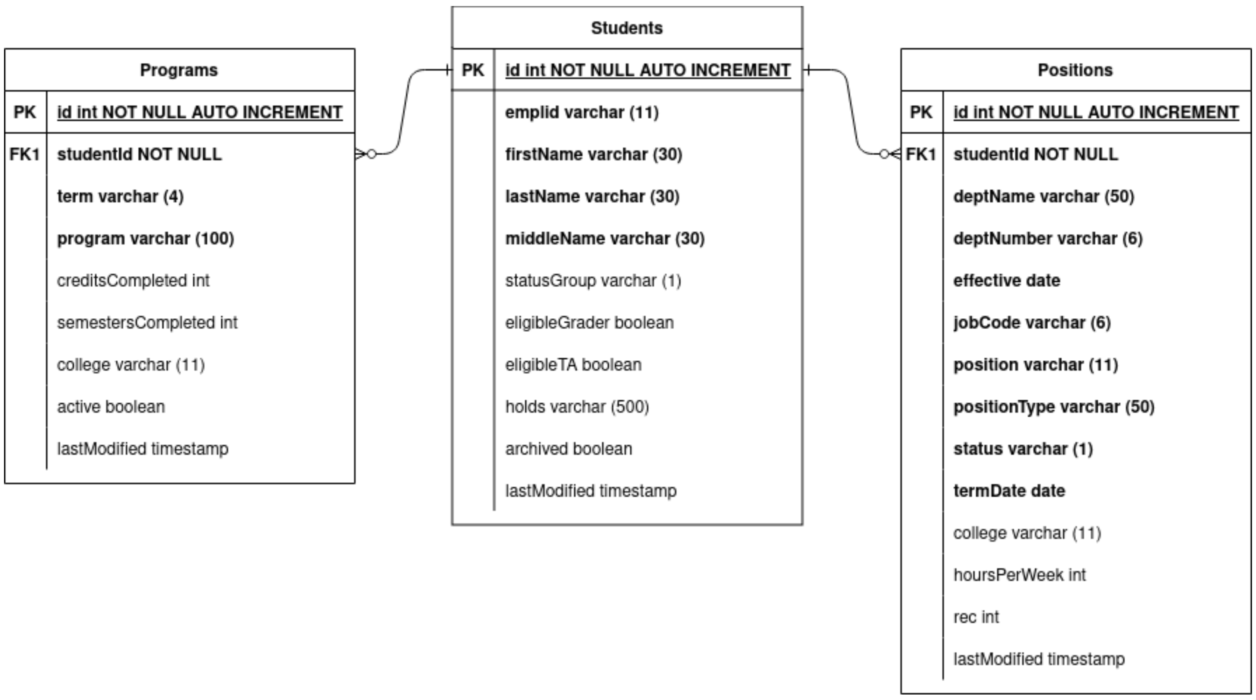
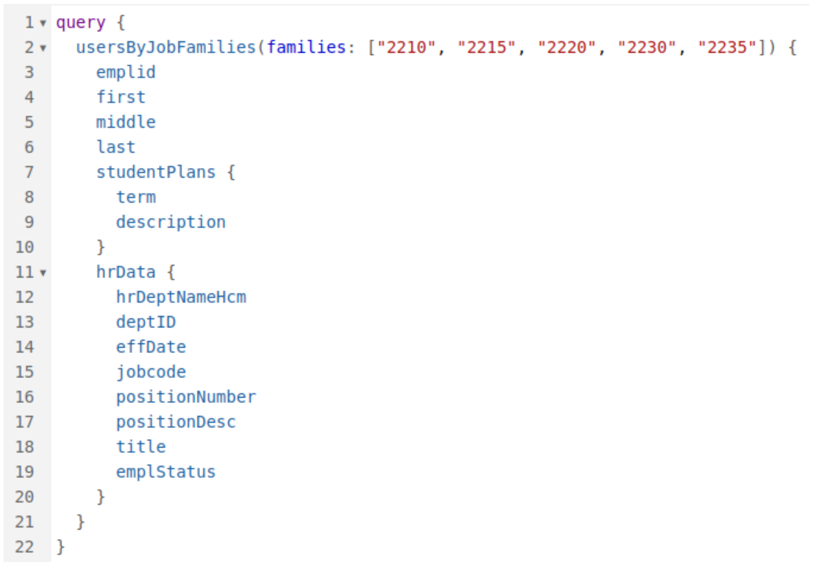


Figure 3.7. Graduate Waiver Wire data diagram for students, programs, and positions. Data automatically updated using a GraphQL query is in bold.

3.3.4. GraphQL Query

To send the query to the Biodemo server, a DGS `DefaultGraphQLClient` is instantiated. This client then executes the query given in Figure 3.8, and its response is fed into a DGS `GraphQLResponse` object. In observing the query, note that the data names called reflect the data as named with the Warehouse biodemo database table (and therefore the GraphQL schema). The names in the Graduate Waivers graduate_warehouse database are different, to reflect what is requested by the Graduate School, though they hold the same data.

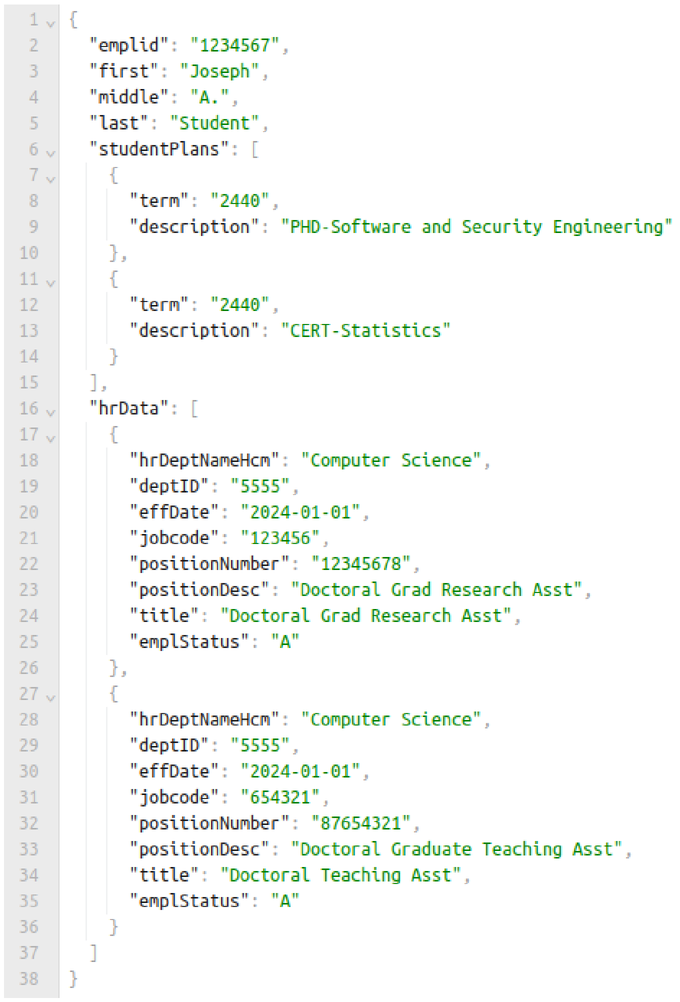
In Figure 3.8, non-graduate-assistants are filtered out in line 2 (as assistantships are categorized by the job families listed), and the request calls for student `emplid`, `first`, `middle`, and `last` name, which will be placed in the students table in graduate_warehouse. The `term` and `description` of all `studentPlans` for that particular student is requested, which will be stored in the programs table. Finally, a number of `hrData` fields are requested, in order to retrieve data for the positions table.



```
 1 query {
 2   usersByJobFamilies(families: ["2210", "2215", "2220", "2230", "2235"]) {
 3     emplid
 4     first
 5     middle
 6     last
 7     studentPlans {
 8       term
 9       description
10     }
11     hrData {
12       hrDeptNameHcm
13       deptID
14       effDate
15       jobcode
16       positionNumber
17       positionDesc
18       title
19       emplStatus
20     }
21   }
22 }
```

Figure 3.8 Graduate Waiver Wire GraphQL query for users (students), plans (programs), and positions (hrData).

To illustrate, Figure 3.9 shows one student as returned from the Biodemo server, with all data falsified to protect student anonymity. The student is in two programs, and they also have two assistantships, illustrating how data containing multiple objects related to one student is delivered.



```
 1 {
 2   "emplid": "1234567",
 3   "first": "Joseph",
 4   "middle": "A.",
 5   "last": "Student",
 6   "studentPlans": [
 7     {
 8       "term": "2440",
 9       "description": "PHD-Software and Security Engineering"
10     },
11     {
12       "term": "2440",
13       "description": "CERT-Statistics"
14     }
15   ],
16   "hrData": [
17     {
18       "hrDeptNameHcm": "Computer Science",
19       "deptID": "5555",
20       "effDate": "2024-01-01",
21       "jobcode": "123456",
22       "positionNumber": "12345678",
23       "positionDesc": "Doctoral Grad Research Asst",
24       "title": "Doctoral Grad Research Asst",
25       "emplStatus": "A"
26     },
27     {
28       "hrDeptNameHcm": "Computer Science",
29       "deptID": "5555",
30       "effDate": "2024-01-01",
31       "jobcode": "654321",
32       "positionNumber": "87654321",
33       "positionDesc": "Doctoral Graduate Teaching Asst",
34       "title": "Doctoral Teaching Asst",
35       "emplStatus": "A"
36     }
37   ]
38 }
```

Figure 3.9. Example student response object.

In Figure 3.9 the user (student) string contains two lists (denoted by square brackets) containing `studentPlans` and `hrData`, which are labeled in lines 6 and 16. It is important to note that this string is one element belonging to a larger user list, where each new user object uses the same format to reflect ownership of one or many `studentPlan` or `hrData` elements.

The next step is to deserialize this large string into data useful within Graduate Waiver Wire.

3.3.5. POJO Classes

GSON offers a number of methods to deserialize JSON, including writing custom deserializers or adding the GSON RuntimeTypeAdapterFactory to theoretically write directly to the database class objects [26]. However, EAD's preferred method of deserializing JSON strings using GSON is to employ basic Java objects (colloquially known as Plain Old Java Objects, or POJOs) to collect the data and pass this along to the Cayenne data model class objects. This is done to provide a layer of separation between the JSON coming in and the Cayenne objects themselves. In addition, for maintenance purposes, a future new Java programmer on the project will easily understand java classes and how GSON is using them. POJO classes contain only fields and getter/setter methods to access them.

Figure 3.10 diagrams the POJOs involved, along with their corresponding Apache Cayenne data model class objects. The fields in the top (POJO) row are directly related to the fields in the bottom (data model) row. Field names within GradUser, HrData, and StudentPlan reflect the names existing in the Biodemo GraphQL server. Field names in the Student, Position, and Program class are slightly different, as they were determined with the assistance of Graduate School personnel during requirements gathering, and allow Apache Tapestry to automatically generate user interface labels usable by the Graduate School.

Within the biodemo service implementation described in Section 3.3.6, a GraphQL response is received and prepared in lines 82-85 of Figure 3.11, then converted into a list of GradUser objects in line 90. Each GradUser object in the array contains its own lists of HrData and StudentPlan objects as parsed from the GraphQL JSON response.

Once the array of GradUser objects is prepared, the update process depicted in Section 3.3.6 incorporates update methods to transfer the data from each GradUser, HrData, and StudentPlan object into their corresponding Student, Position, and Plan Apache Cayenne data model objects. An example update method, to update a Student based on a GradUser parameter, is shown in Figure 3.12, in which the Student values are set using the values contained in the GradUser parameter:

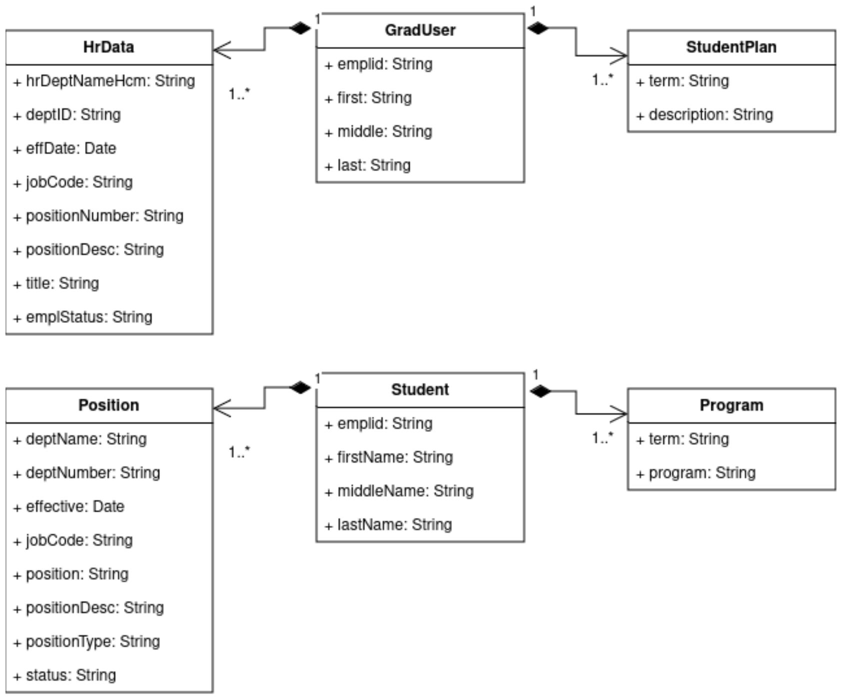
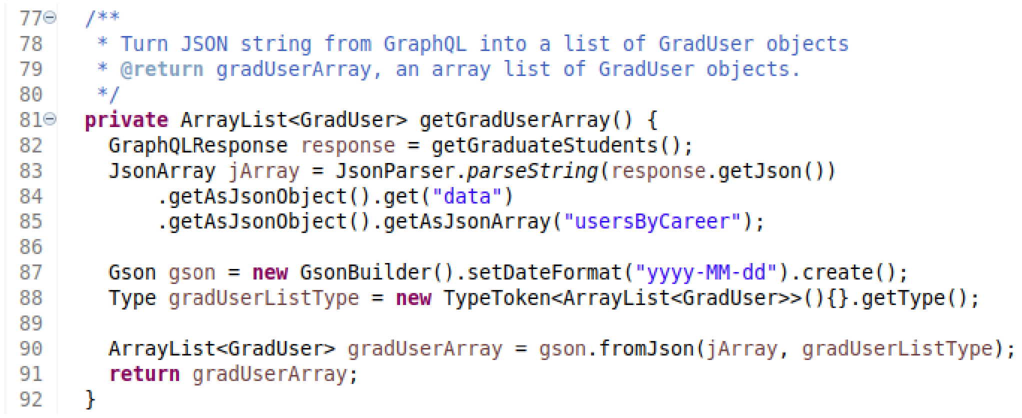
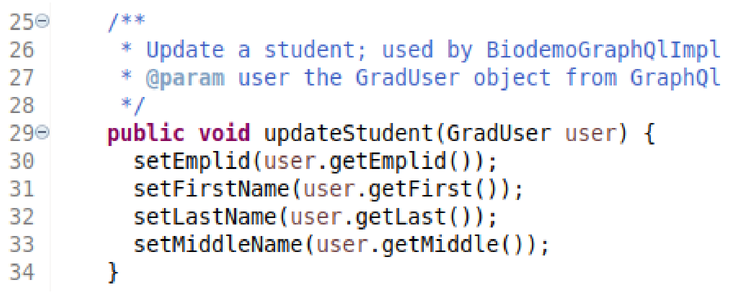


Figure 3.10. Plain Old Java Objects for JSON deserialization (top), and their corresponding Apache Cayenne data model classes (bottom). A GradUser has one-to-many relationships with HrData and StudentPlan objects; the Student class correspondingly has one-to-many relationships with the Position and Program class.



```
/**
 * Turn JSON string from GraphQL into a list of GradUser objects
 * @return gradUserArray, an array list of GradUser objects.
 */
private ArrayList<GradUser> getGradUserArray() {
  GraphQLResponse response = getGraduateStudents();
  JsonArray jArray = JsonParser.parseString(response.getJson())
      .getAsJsonObject().get("data")
      .getAsJsonObject().getAsJsonArray("usersByCareer");

  Gson gson = new GsonBuilder().setDateFormat("yyyy-MM-dd").create();
  Type gradUserListType = new TypeToken<ArrayList<GradUser>>(){}.getType();

  ArrayList<GradUser> gradUserArray = gson.fromJson(jArray, gradUserListType);
  return gradUserArray;
}
```

Figure 3.11. Deserializing JSON into a list of GradUser objects.



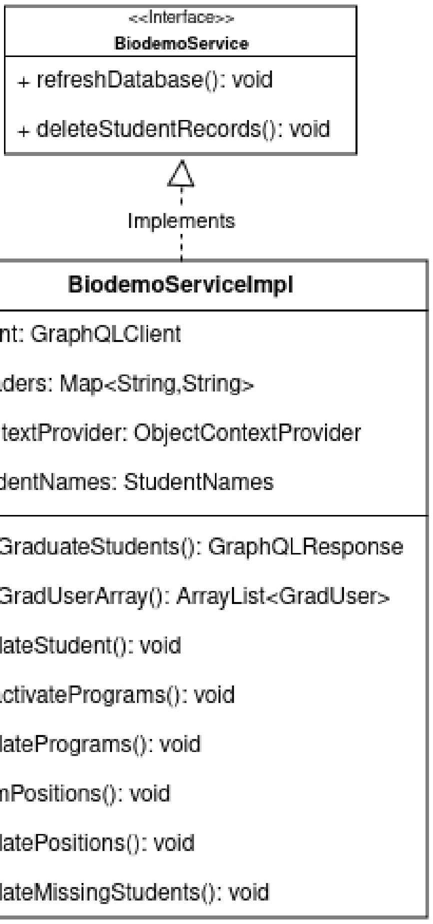
```
    /**
     * Update a student; used by BiodemoGraphQlImpl
     * @param user the GradUser object from GraphQl
     */
    public void updateStudent(GradUser user) {
      setEmplid(user.getEmplid());
      setFirstName(user.getFirst());
      setLastName(user.getLast());
      setMiddleName(user.getMiddle());
    }
```

Figure 3.12. Method to update a Student based on a GradUser object.

Similar methods to update a Position (based on a HrData parameter) and a Program (based on a StudentPlan parameter) exist to serve the same purpose: passing data to the Apache Cayenne objects so they can be committed to the graduate_warehouse database.

3.3.6. Biodemo Service Implementation

Within the Tapestry Application, the GraphQL client runs within a single class, BiodemoServiceImpl, which is an implementation of a BiodemoService interface and is diagrammed in Figure 3.13.



<<Interface>>
BiodemoService
+ refreshDatabase(): void
+ deleteStudentRecords(): void
Implements
BiodemoServiceImpl
+ client: GraphQLClient
+ headers: Map<String,String>
+ contextProvider: ObjectContextProvider
+ studentNames: StudentNames
+ getGraduateStudents(): GraphQLResponse
+ getGradUserArray(): ArrayList<GradUser>
+ updateStudent(): void
+ deactivatePrograms(): void
+ updatePrograms(): void
+ termPositions(): void
+ updatePositions(): void
+ updateMissingStudents(): void

Figure 3.13. BiodemoServiceImpl class.

EAD's Tapestry Archetype is designed to implement recurring jobs based on service interfaces, which is why this structure is chosen. The methods given in Figure 3.13 are utilized in the refreshDatabase method, which is described in Algorithm 1.

```
Algorithm 1: Refresh Database
Result: Updated students, programs, and positions
create Cayenne data context;
create maps of existing students, programs, positions;
get GradUser array (GraphQL)
for gradUser in GradUsers do
    create student;
    if studentMap contains gradUser then
        student = get student from studentMap
    else
        student = new data context object
    end
    get existing programs list;
    for studentPlan in gradUser studentPlans do
        create program;
        if programMap contains studentPlan then
            program = get program from programMap;
            remove program from existing programs list
        else
            program = new data context object
        end
        update program;
    end
    get existing positions list;
    for hrData in gradUser hrData do
        create position;
        if positionMap contains hrData then
            position = get position from positionMap;
            remove position from existing positions list
        else
            position = new data context object
        end
        update position;
    end
    update student;
    set remaining programs in existing list to deactivated status;
    set remaining positions in existing list to terminated status;
end
update missing students;
```

As Algorithm 1 depicts, a GraphQL client sends the request to the server, along with a header containing the security key in getGraduateStudents, which returns the JSON response, which is then turned into an array of GradUser objects, each containing its own set of StudentPlan and HrData objects. For each GradUser object, either an existing, corresponding Student object is selected for update or a new Student is created. For each StudentPlan and HrData element owned by the GradUser, the process is repeated (either update existing or create new). For each Student and their Programs and Positions, any changes are saved to the graduate_warehouse database. Once this is completed, if any existing students, programs, or positions have not shown up in the current feed, they are respectively set to missing, deactivated, or terminated.

This process is scheduled to run once every twelve hours through Apache Tapestry's AppModule class, which allows services to be automated. This provides Graduate Waiver Wire with twice-daily updates of Student, Program, and Position data.

4. DISCUSSION AND CONCLUSION

4.1. Discussion

The Graduate Waiver Wire application currently completes successful updates twice daily using GraphQL with no issues. It should be noted that the number of Graduate Students fetched hovers around 1,000. Applications accessing larger data sets may encounter more N+1 issues, as discussed in Section 3.2.5.

Graduate School personnel have reported many benefits from Graduate Waiver Wire and its automated updates. GraphQL automation frees the time spent by Graduate School personnel in manually tracking new students, positions, and programs, as was the case when using the old Microsoft Access file. All information for each student is now in one central location, and automation means new students are added seamlessly into the application. This automated basis combined with manual entry has allowed for improved response times to student questions, and for the addition of new features such as report generation. It has also helped prevent costly human errors such as awarding a waiver to an unqualified student.

4.1.1. Update Time

To get a sense of update times, tests were run that focused on the time GraphQL took to return a set of results, and the time it took for the BioDemo Service Implementation class to update the database with the results. The data returned by the GraphQL JSON response consisted of 984 users (students in Graduate Waivers), 2,369 studentPlans (programs), and 1,213 hrData (positions) objects. For the Update Database evaluation, the database was cleared of all data prior to each run, to simulate an update of the entire data set (in the actual application, only a small subset of the full data set is updated each time). Tests were run on a desktop computer using an Intel(R) Core(TM) i5-9600 CPU with a max speed of 4.6 gHz, and 16 MB RAM.

In twenty GraphQL calls, the response time varied from 0.498 seconds to 1.784 seconds, with a mean of 0.797 seconds as shown in Table 4.1. Updating the database from the JSON response took longer, varying from 1.225 to 3.427 seconds, with a mean of 2.417 seconds. The increased running time of the Update Database function is expected, as the algorithm contains nested loops. In further work, the refresh database method may benefit from further analysis and refactoring in order to improve its speed.

Table 4.1. GraphQL Response and Refresh Database Times in Seconds.

Function	Mean	Median	All times
GraphQL Response	0.797	0.574	0.922, 1.784, 0.837, 0.498, 0.565, 0.505, 0.533, 0.537, 0.555, 0.512, 0.588, 0.533, 0.720, 0.582, 0.497, 1.324, 1.114, 1.483, 1.282, 0.565
Update Database	2.417	2.444	2.354, 2.691, 1.711, 2.325, 3.343, 1.335, 1.728, 2.573, 2.365, 2.844, 1.704, 3.158, 2.382, 2.505, 3.427, 2.913, 3.142, 1.487, 1.225, 3.137

4.1.2. Benefits Related to GraphQL

From a developer perspective, the main benefit of GraphQL is rapid development of the application GraphQL client. The JSON-like format of a GraphQL request, which lists the objects needed, their related sub-objects, and only the fields required by that application, is very easy to understand and implement. In addition, Gson deserialization tied to POJO classes is a very straightforward process. This allows for improved collaboration and maintenance, as developers have little trouble joining work on applications which may be new to them, but use GraphQL to fetch data.

4.1.3. Drawbacks of GraphQL

From an EAD perspective, there is a steeper learning curve in working server-side. EAD has implemented its service using Java/Groovy to connect to a MySQL database, but in any programming setup the main challenges will be in understanding how the schema relates to the data classes, and how these work with the resolver classes to interact with the database. The upside is a service only needs to be implemented once in order to be accessed by many clients.

The other downside is the potential for inefficient queries due to the N+1 problem. Smaller datasets appear to have few problems simply due to their small size, but accessing larger complex datasets will require mitigation strategies.

4.2. Conclusion

This paper has discussed how EAD uses GraphQL to provide applications access to data. The ability to make complex requests in a single query, alongside the ability to fetch only what data is needed, have been especially useful. In addition to Graduate Waiver Wire, a number of other applications have begun using GraphQL and further application development will continue to do so, especially as development on the server side has reached a state of maturity.

Larger institutions or those planning on querying much larger data sets should consider how to best mitigate query inefficiency, but for similar-sized universities that develop specialized applications, GraphQL may provide a number of benefits, as it has done for North Dakota State University's EAD team.

REFERENCES

[1] *GraphQL vs REST API - Difference Between API Design Architectures - AWS*. (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/compare/the-difference-between-graphql-and-rest/

[2] *The Apache Groovy programming language - Working with a relational database*. (n.d.). Groovy-Lang.org. https://groovy-lang.org/databases.html

[3] *Apache Cayenne*. (n.d.). Cayenne.apache.org. https://cayenne.apache.org

[4] *GraphQL: A query language for APIs*. (n.d.). Graphql.org. https://graphql.org/learn/

[5] *GraphQL Specification Versions*. (n.d.). Spec.graphql.org. http://spec.graphql.org/

[6] *GraphQL Code Libraries, Tools and Services*. (n.d.). Graphql.org. https://graphql.org/code/

[7] Gudabayev, T. (2021, October 20). *A brief history of graphql*. DEV Community. https://dev.to/tamerlang/a-brief-history-of-graphql-2jhd

[8] *Frequently Asked Questions (FAQ) | GraphQL. (n.d.)*. Graphql.org. https://graphql.org/faq/

[9] *Who's Using | GraphQL. (n.d.)*. Graphql.org. https://graphql.org/users/

[10] Srinivasan, K. (2023, August 14). *Evolving the federated GraphQL platform at Netflix*. InfoQ. https://www.infoq.com/articles/federated-GraphQL-platform-Netflix/

[11] Bryant, M. (2017, December). GraphQL for archival metadata: An overview of the EHRI GraphQL API. In *2017 IEEE International Conference on Big Data (Big Data)* (pp. 2225-2230). IEEE.

[12] Ulrich, H., Kern, J., Tas, D., Kock-Schoppenhauer, A. K., Ückert, F., Ingenerf, J., & Lablans, M. (2019). QL4MDR: a GraphQL query language for ISO 11179-based metadata repositories. *BMC medical informatics and decision making*, 19(1), 1-7.

[13] Haris, M., Farfar, K. E., Stocker, M., & Auer, S. (2021, November). Federating scholarly infrastructures with GraphQL. In *International Conference on Asian Digital Libraries* (pp. 308-324). Cham: Springer International Publishing.

[14] Farré, C., Varga, J., & Almar, R. (2019). GraphQL schema generation for data-intensive web APIs. In *Model and Data Engineering: 9th International Conference, MEDI 2019,* Toulouse, France, October 28–31, 2019, Proceedings 9 (pp. 184-194). Springer International Publishing.

[15] Taelman, R., Vander Sande, M., & Verborgh, R. (2018). GraphQL-LD: linked data querying with GraphQL. In *ISWC2018, the 17th International Semantic Web Conference* (pp. 1-4).

[16] Hartina, D. A., Lawi, A., & Panggabean, B. L. E. (2018, November). Performance analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University. In *2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT)* (pp. 237-240). IEEE.

[17] Brito, G., & Valente, M. T. (2020, March). REST vs GraphQL: A controlled experiment. In *2020 IEEE international conference on software architecture (ICSA)* (pp. 81-91). IEEE.

[18] Vadlamani, S. L., Emdon, B., Arts, J., & Baysal, O. (2021, June). Can graphql replace rest? a study of their efficiency and viability. In *2021 IEEE/ACM 8th International*

Workshop on Software Engineering Research and Industrial Practice (SER&IP) (pp. 10-17). IEEE.

[19] Hartig, O., & Pérez, J. (2017). *An initial analysis of Facebook's GraphQL language*. https://repositorio.uchile.cl/xmlui/bitstream/handle/2250/169110/An-initial-analysis-of-facebooks-GraphQL-language.pdf?sequence=1

[20] Hartig, O., & Pérez, J. (2018, April). Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference* (pp. 1155-1164).

[21] *DGS Framework*. (n.d.). Netflix.github.io. https://netflix.github.io/dgs/

[22] *graphql-java/graphql-java*. (n.d.). GitHub. https://github.com/graphql-java/graphql-java

[23] Roksela, P., Konieczny, M., & Zielinski, S. (2020, July). Evaluating execution strategies of GraphQL queries. In *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)* (pp. 640-644). IEEE.

[24] Robinson, Ed (2023, December 10). *Introduction to the GraphQL N + 1 Problem*. Caisy.io. Retrieved March 14, 2024, from https://caisy.io/blog/understanding-solving-graphql-n-1

[25] *Gson User Guide* (n.d.). https://github.com/google/gson/blob/main/UserGuide.md

[26] Paraschiv, E. (2022, June 24). *Gson Deserialization cookbook*. Baeldung. https://www.baeldung.com/gson-deserialization-guide