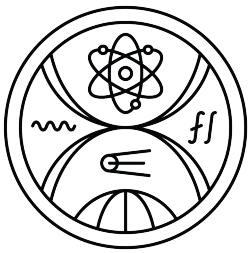
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND
INFORMATICS

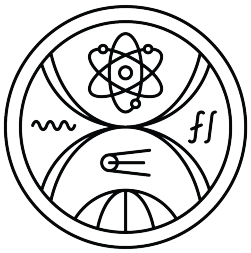


ACCESS CONTROL FOR CLIENT-SIDE GRAPH-BASED QUERIES

Diploma thesis

2023 Bc. Miroslav Baluch

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND
INFORMATICS



ACCESS CONTROL FOR CLIENT-SIDE GRAPH-BASED QUERIES

Diploma thesis

Study programme:	Applied Computer Science
Field of Study:	Computer Science
Department:	Department of Applied Informatics
Supervisor:	Mgr. Ján Kľuka, PhD.

Bratislava, 2023 Bc. Miroslav Baluch

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Miroslav Baluch
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Access control for client-side graph-based queries

Annotation: Modern full-stack web development approach favours data access via graph-base query languages such as GraphQL and JSON-LD-Query. Client-side query composition offers flexibility for the developers but introduces security risks as queries composed on the client are executed on the server.

Aim: The goal is to propose an access control mechanisms that enables to verify and pass only such data to the client for which the current user actually has access. This will be implemented and tested in the existing system courses.matfyz.sk which uses JSON-LD data representation.The goal of this thesis is to propose an access control mechanisms that enables to verify and pass only such data to the client for which the current user on the client side has permissions. This will be implemented and tested in the existing system courses.matfyz.sk which uses JSON-LD data representation.

Literature:

1. Homola, M., Kl'uka, J., Kubincová, Z., Marmanová, P. and Cifra, M., 2019. Timing the Adaptive Learning Process with Events Ontology. In International Conference on Web-Based Learning (pp. 3-14). Springer.
2. Antoniou, G. and Van Harmelen, F., 2004. A semantic web primer. MIT press.
3. Taelman, R., Vander Sande, M. and Verborgh, R., 2019. Bridges between GraphQL and RDF. In W3C Workshop on Web Standardization for Graph Data. W3C.
4. Taelman, R., Vander Sande, M. and Verborgh, R., 2018. GraphQL-LD: Linked Data Querying with GraphQL. In International Semantic Web Conference (P&D/Industry/BlueSky).

Supervisor:	Mgr. Ján Kľuka, PhD.
Consultant:	doc. RNDr. Martin Homola, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	21.10.2018

Approved: 13.10.2021 prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

.....

Student

.....

Supervisor

I declare honestly that I have developed this diploma thesis separately only using the literature and with the help of my supervisor.

.....

Bratislava, 2023 Bc. Miroslav Baluch

Thanks

TBD

Abstract

The work deals with the design and implementation of control of access rights of graph requests on the client side for the support study system named Courses. The current system does not sufficiently control access rights, with result that unauthorized users having access to data to which they should not have access. Due to this fact this system which is already fully deployed and used by students of the Faculty of Mathematics, Physics, and Informatics, will be modified and expanded with new components. The main changes will be made to the current application server, which ensures the operation of the application and provides users with the necessary and optional data. As this server is currently not sufficiently protected, each user has almost full access to the entire database. In addition, HTTP requests to the application server currently retrieve all data, even that the user does not need at all. This results in higher application and database server utilization, slower user response, and higher data transfer. Due to upcoming changes in the application server, it will be possible to extract a certain part of data and send only those which are required by the client. A common interface will be done for requests which will serve as the frontend part of the application. An additional feature that should be added is the simplification of the creation and deployment of new instances of the course system. In addition, straightforward changes in the database structure will be processed by the application server without the

need for intervention by the programmer.

Keywords: application server, database, ontology, access rights, RDF

Abstrakt

Práca sa zaoberá navrhnutím a implementovaním kontroly prístupových práv grafových dopytov na strane klienta pre podporný kurzový systém Courses. Momentálny systém nedostatočne kontroluje prístupové práva, čo ma za následok, že neautorizovaný používatelia majú prístup k dátam, ku ktorým by prístup nemali mať. Kvôli tomu bude tento systém, ktorý už je plne nasadený a využívaný študentami Fakulty matematiky, fyziky a informatiky pozmenený a rozšírený o nové súčasti. Hlavnou zmenou prejde momentálny aplikačný server, ktorý zabezpečuje fungovanie aplikácie a podáva používateľom potrebné dáta. Keďže tento server nie je momentálne dostatočne chránený, každý používateľ má prístup takmer k celej databáze. Okrem toho sa momentálne pomocou HTTP požiadaviek na aplikačný server získavajú všetky dáta, aj tie, ktoré používateľ vôbec nepotrebuje. To ma za dôsledok vyššiu vyťaženosť aplikačného a databázového servera, pomalšiu odozvu pre používateľov a vyšší prenos dát. Vďaka zmenám v aplikačnom serveri bude možnosť vytiahnuť iba určitú skupinu dát. Taktiež je snahou zjednodušiť vytváranie a nasadenie nových inštancií tohto kurzového systému. Zároveň jednoduchšie zmeny v štruktúre databázy budú spracované aplikačným serverom bez nutnosti zásahu programátorom.

Kľúčové slová: aplikačný server, databáza, ontológia, prístupové práva, RDF

Contents

Introduction

Education is a crucial aspect of this century characterized by rapid development, with major changes in every field of study. Web tools are not an exception and are created every moment to catch up with the trend. New services rise with a focus on the simplification of the teaching process. Web applications are in such growth because of their easy accessibility, with almost instantaneous availability and mostly they do not require the user to own the most recent hardware. In this thesis, I also strive to create a web application server that will be used as the main backbone for the new student system.

My work will be based on the existing diploma thesis of Milana Cifru, Semantic Data Model for a Course Management System [Cif20].

I will work with the code of this existing thesis and use it to expand the functionality of the back-end part of the application. The current back end can handle requests from the client, fulfill them and send back the response. The application can handle simple requests, while complex ones are also supported. Custom routes and HTTP methods are also used in the system to give the front-end developers more possibilities. The application server communicates with the database server and supports the insertion, update, and deletion of the information. There is no common interface made for front-end developers when sending requests. Each developer invented his

way of how the requests are created, thus making the debugging process in case of issues harder and the code more is complex. The access management system is also present in the application, thus its usage is quite limited and must be improved to increase the security measure.

The database server will not be changed and I will use an RDF database server called Virtuoso. Virtuoso is flexible and connected - we can easily find out concrete attributes, e.g. User, or find out all User's first names with ease.

The main priority is to be able to query and modify data over the RDF database. For this case, I will use a GraphQL interface. Two options could have been used for this matter. The first option (HyperGraphQL) however does not support modification and extraction of schema therefore we went with a second option called UltraGraphQL.

The UltraGraphQL has to run over our RDF triple store database (Virtuoso server). It will provide an easy way to access the data in the database. However, for UltraGraphQL to be able to fetch data, we need a complete schema. Normally UltraGraphQL doesn't see the schema so it must be provided to it and thus an export script (which we call Create Script) must be created. The schema was never fully described in RDF data, but in the backend, it is written in JavaScript form as a JavaScript object. The schema will be extracted from the backend code and after that served to UltraGraphQLEndpoint. Furthermore, Create Script provides us with an easy way how to create additional instances (for example for testing) which can be even automatized.

Chapter 1

Semantic Data Model

1.1 Linked Data

Linked Data [BHBL11] represents data that are available on the web. They are stored in machine-readable form. Data are interlinked and are represented by entities. Each entity can have an indefinite number of attributes. Attributes describe properties of the given entity. Entities can be found by attributes that are assigned to them or list all attributes for the given entity. Each entity has an IRI, an Internationalized Resource Identifier which symbolizes the name or the identifier of the given entity. Usually, they are similar to classic URLs which are used to access web pages. Indeed they are similar as each URL is an IRI, but not each IRI is an URL because of special characters which can be also used in the IRIs. IRI can represent anything, define a person from the real world, an object, or some fictional items - there is no limit to what it can represent, it is just an identifier for something with attributes that can provide more information about the given entity. Each attribute is also an identifier which can have other attributes. Links between entities are described by RDF and are encoded as HTTP URIs. This al-

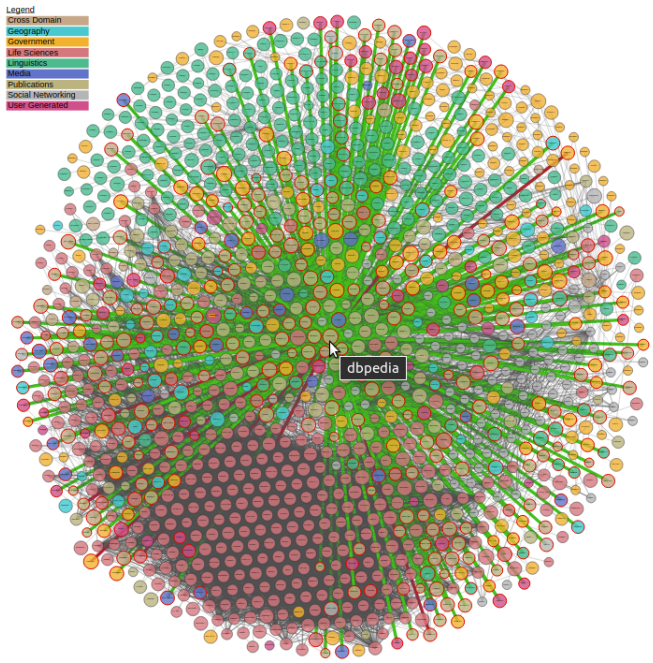


Figure 1.1: Links between datasets represented as graph nodes from [Wik21]

lows the content to be accessible in a readable way for humans and also for machines.

Entities are located on the given domain and can be accessed from this domain by the identifier. The domain represents a dataset. Each dataset can be then connected to another dataset. By multiple datasets, we are creating a network of connected datasets which usually are also interlinked. Linked data are usually represented in a graph way, where datasets and entities are presented as nodes that are linked.

1.2 Graph representation

Graph representation use graph structures to represent the structure of data. Nodes represent entities and each entity can have multiple multiple attributes and relations with other nodes. Attributes and relations are represented by

links (or edges in the graph sense). Graphs used in this sense are oriented, therefore the relation or attributes goes from one node to another but might not go the other way.

1.3 Resource Description Framework (RDF)

Resource Description Framework [SARB14] is used to describe data (entities) and information about them (connections between them). Its mainly used for applications and processes which should be capable to read the information. Description can be made for any object, or record, including individuals or some conceptual ideas. The syntax is commonly based on XML, Terse RDF Triple Language (Turtle), or JavaScript Object Notation for Linked Data (JSON-LD).

The description of entities is made in form of triples. Each triple represents a statement or information about the given entity. The first parameter of the triple is a subject or entity about which we are creating a statement. The second parameter is a predicate which tells what we want to describe the given relationship between subject and object. A third parameter is an object which represents another entity, which we interlinked with the subject. The statement (or relation) for one triple is always defined from subject to object and not in the other way.

```
<subject> <predicate> <object>
```

Listing 1.1: Notation of statement as triple

1.3.1 Structure of triple

An entity represents an IRI, an identifier. IRI is an object, a record, a person, or some conceptual idea. The subject of the triple must be always an IRI, while the object can be an IRI or some data-type object (called literal). Literal is plain values that do not represent an IRI. Literal can be any data types, like string, date, or number. When we supply a literal to an object we always associate the respecting datatype so that the value can be properly processed by machines.

Statements can be made about any IRI. Even if the IRI does not have any statements about it, the IRI exists and a statement can be made about it in the future. We call IRIs that does not have any statements about them as blank nodes. These nodes in the graph representation do not have any edges and are not linked with any other nodes, however, they can appear in the subject or object position of the triple.

```
<John> <is a> <User>.
<John> <is a student of> <Mathematics 2022/23>.
<John> <nickname is> <Legolas>.
<John> <has public profile> <True>.
<John> <is a member of> <Ninjas>.
```

Listing 1.2: Statements about entity

In the examples 1.2, we made some statements about John, the subject. John also represents an IRI, an identifier. As we already know, a statement consists of a triple. The first argument in our case is John. The second argument of the triple is reserved for the predicate that implies what we want to tell about John and in which relation he is with the object. The third argument, the object, tells us that John is specifically in relation to

this object.

Examples represent the existing MatFyz Courses system structure of Users and Teams and show relations between multiple models. The MatFyz Courses system's design is described in depth in chapter 3.

In the first example, we are saying that John is a User. This statement is also human-readable. A User, in this matter, might present a literal (a data type) or some other IRI. Usually, we do not mark these types as literal but as IRIs, which we call classes. Classes define some information about the given object in a more general matter. In the following example, we are stating that John is a student of Mathematics 2022/23. We are linking these two IRIs together, creating a relationship between John and Mathematics 2022/23. There might also be some statements about Mathematics 2022/23. In the third example, we assign to John his nickname. A nickname, the subject is, in this case, a string that we represent as a literal - a data type. The last example tells us that John is a member of another entity identified as Ninjas.

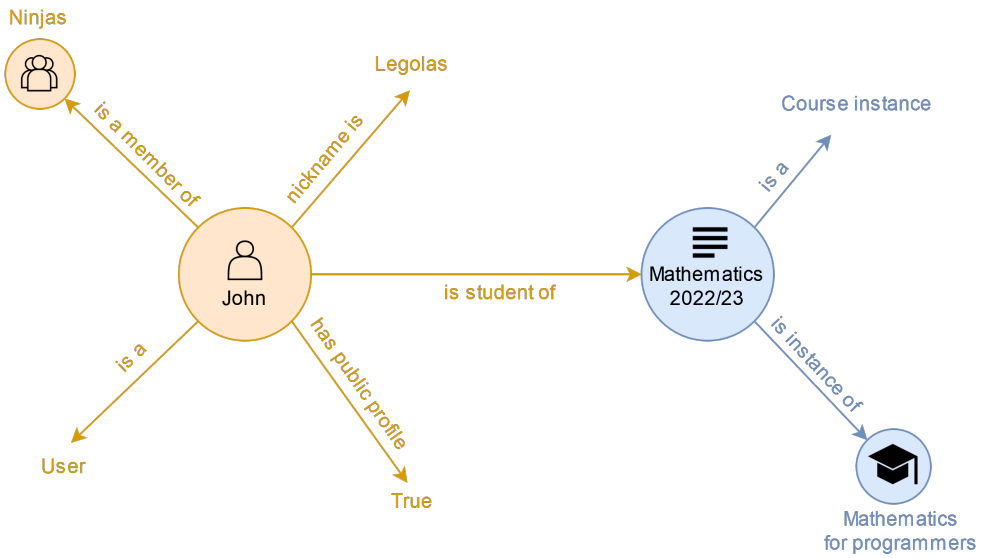


Figure 1.2: An example of entities with relations and attributes.

1.3.2 Turtle Syntax

The statements in the example 1.2 abstractly represent the triples. Therefore, we can use one of the possible syntaxes to write the triples. Each syntax has its own syntax rules. One of the most common is Turtle syntax. They represent an extension of N-Triples, which are just plain triples. Turtle syntax helps to make the notation straightforward by providing syntax shortcuts, like defining prefixes so that we don't have to type relative or absolute IRIs for each triple.

```
@PREFIX courses: <http://www.courses.matfyz.sk/> .
@PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@PREFIX schema: <http://schema.org/> .
@PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> .

courses:User a rdfs:Class ;

courses:John
    a courses:User ;
    courses:studentOf courses:Mathematics 2022/23 ;
    courses:publicProfile "false"^^xsd:boolean ;
    courses:nickName "Legolas" ;
    course:memberOf courses:Ninjas ;

courses:Mathematics 2022/23
    a courses:CourseInstance ;
    courses:instanceOf courses:Mathematics for programmers ;
```

Listing 1.3: Notation in Turtle syntax

Each prefix starts with a '@' character. With prefixes, we are making pointers to ontologies. After that, we can begin typing triples from 1.2. So instead of absolute IRI, we use the prefix of the given ontology, followed by a predicate. If we have multiple triples about the same subject, we don't have to type the subject for each triple. Instead, we write a subject in one line and add the predicate and object in additional lines. In this case, each statement ends with the ';' character. When we look at the first example, it uses the predicate 'a', a short notation for rdf:type, which is taken from the RDF schema.

1.4 Data-modelling vocabulary

Triples define relationships between entities. For a better understanding of these relationships, it's beneficial to assign them a semantic meaning. We can create our semantic vocabulary or use the existing one. For some standard types, it is better to use the global semantics that is already commonly used. With that, we ensure that the meaning of these relationships will be identical when we create them, and other ontologies will easily recognize them. Semantic vocabularies in RDF used to describe the relationships between entities are called schemas.

1.4.1 RDF schema

RDF schema mainly consists of class and property systems.

The most basic notation of the RDF schema is an rdfs:Resource. Therefore, each entity written by the RDF is also defined as a resource. One of the most commonly used notations is an rdfs:Class. It helps us to determine the category of the RDF entity. Class meaning is similar to the one used in

object-oriented programming languages like Java. The statement that some subject type is a Class means that when we create an object with this type of class, we know that it's an instance of the given class.

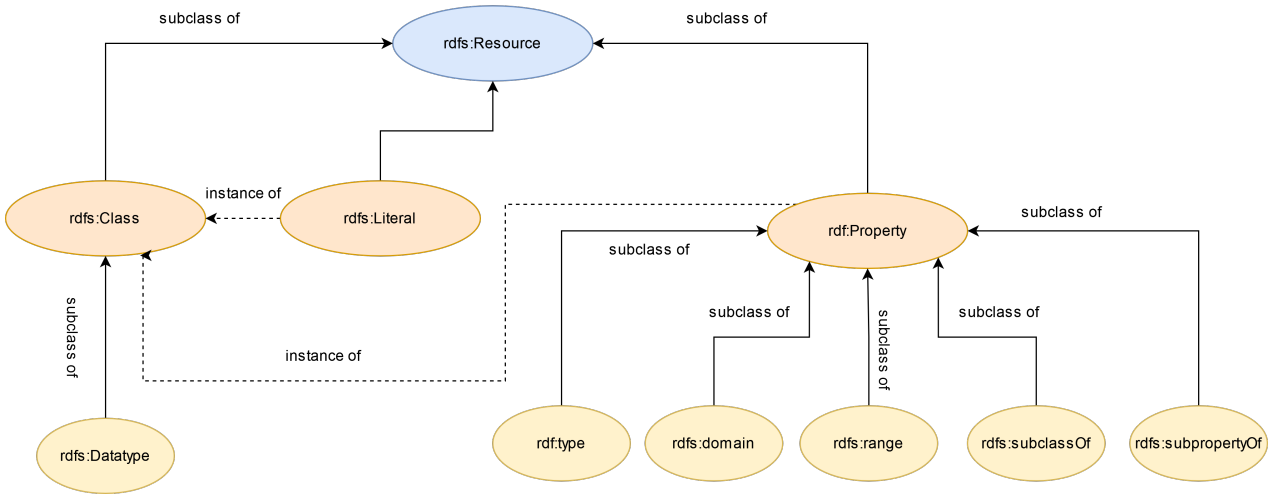


Figure 1.3: Common used notations in RDF schema.

Other commonly used notations are rdfs: Literal and rdfs:Property. All of them are subtypes of the rdfs:Resource type. Each subject that is entitled as rdfs:Literal represents a literal. A literal represents string, integer, or other data type. Therefore, we usually assign a notation rdfs: Datatype to these subjects and a subtype of rdfs: Literal.

Notation rdfs:Property defines a relation between subject and object resources. There exist multiple instances of this notation, but these are the most commonly used:

With notations rdfs:range and rdfs:domain we describe what can be used on the subject's or the object's side - field of values and the domain. The field of values is described through rdfs:range - that the values, subjects, are instances of one or more specified classes. The domain defined by the rdfs:domain means that any resource, the object, is an instance of one or

more classes.

To state that a given resource belongs to the instance of a class, we use the notation rdf:type or, in short, the notation 'a'.

If we want to define that one class is a subtype of another class, we use the notation rdfs:subClassOf. This notation is also transitive, meaning that if A is a class and is a subclass of B and B is a class and a subclass of another class C, then A is a subclass of C.

The same things can also be defined for the property - one property can be a subproperty of another. This notation is also transitive.

1.4.2 OWL

OWL [HKP+12] is used to describe ontologies. It provides rich notations and streamlines the process of creating new ontologies. Owl can be written in multiple syntaxes, like Turtle, RDF/XML, or OWL2 XML syntax. It is an extension of RDFS notations. For example, a notation owl:ObjectProperty is a subclass of the RDF class rdf:Property. Notations are usually the same or similar to the ones used in the RDFS. However, it adds new attributes and properties. For sets, we can define union, intersection, or conjunction. For example, rdfs:domain should be represented as a conjunction as we should be able to explain one or multiple subjects belonging to the class. Thus, in that case, we should use owl:unionOf. For example, we have an attribute name that should be a part of a Team and Course. In this case, we might write:

```
<owl:ObjectProperty rdf:ID="name">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
```

```
            <owl:Class rdf:about="#Team"/>
            <owl:Class rdf:about="#Course"/>
          </owl:unionOf>
        </owl:Class>
      </rdfs:domain>
</owl:ObjectProperty>
```

Listing 1.4: Domain of attribute name is Team and Course

1.5 Representation in JSON-LD

The JSON for Linked Data represents [SARB14] linked data using the JSON format. It provides an easy change of JSON documents to RDF. Developers can effortlessly write in this format as it is human-readable and writable. On the web, it is, for example, used with the REST API. Each object defined through JSON-LD must contain required fields, like id. Field id starts with the character '@' followed by the object's identifier. Field @id can be an object identifier or a reference to an object identifier across multiple JSON files.

1.6 SPARQL

SPARQL [PS08] is a query language used for data written in the RDF format. When we write a query, it is run over the database and returns an output shown in one of the possible syntaxes of the RDF. The basic actions in SPARQL are SELECT, INSERT and DELETE actions. The syntax of the queries is similar to the one used in relational databases. The query can return a triple or part of the triple. SELECT will return data in the graph

that matches the given condition.

```
@PREFIX courses: <http://www.courses.matfyz.sk/> .
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

SELECT ?user
WHERE {
?user a courses:User .
}
```

Listing 1.5: SELECT query for finding subjects that are of type User

INSERT is used to insert data into the database in the specific graph. A graph is similar to a relational database, which has multiple tables. However, there are no tables in the RDF databases as in the relational databases. Instead, we have a graph containing all of its specified triples. If the graph is not specified in the query, then the default one is used for querying. DELETE action serves for the deletion of specific triples from the graph.

```
@PREFIX courses: <http://www.courses.matfyz.sk/> .
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

INSERT DATA { GRAPH <https://courses.matfyz.sk> {
        courses:John a courses:User .
        courses:John courses:nickName "Legolas" .
 }
}
```

Listing 1.6: INSERT query to the specific graph

Specific notations make it possible to perform INSERT and DELETE as one action (as one transaction), which in the end serves as an UPDATE action. We can also perform an ASK query, which does not return a triple,

but a simple answer, True or False, for the given matter.

1.7 Virtuoso

Virtuoso [Sof20] is a database engine that supports the querying of RDF data through SPARQL. In the sense of RDF, it consists of multiple graphs. The web interface is used to configure the Virtuoso server, providing management of the Virtuoso users, rules, and other configs. The query editor is also bundled in the basic version. Users execute queries. Each user has a set of rules defining which actions might be executed. Rules can be defined for the whole database server or the specific graph. Two versions of Virtuoso exist. One is a free and open source edition, and paid one.

Chapter 2

GraphQL

This chapter is devoted to the explanation of the GraphQL specification and its superstructures.

2.1 GraphQL interface

2.2 HyperGraphQL

2.3 UltraGraphQL

Chapter 3

Existing Course System

This chapter explains the structure and functionality of the existing application, which is already deployed and used. Understanding the existing backend is a critical part of creating additional features.

3.1 Base technologies

This diploma thesis will use standard technologies that are common for the creation of the application server.

This includes JavaScript, Node.js, NPM, React, Apollo Client and Java. JavaScript [con22b] is a scriptable interpreted programming language. It is used primarily in the creation of interactive website pages or for application servers. The application server used in this thesis is based on the Express.js, Node.js web application framework.

Node.js [con22a] is used for application servers, which represents the server side of the web application. It is for example used to take requests from the front-end side of the application, fulfill them and then send a response back to the front end. It is not bound to the browser and is run

directly on the given OS server (or computer).

However, writing specific handlings of the requests in plain Node.js might be sometimes cumbersome. For that, there exist web frameworks that make the handling of requests easier, with more options without a need to reinvent the wheel.

One of the most commonly used is Express.js which is built on top of Node.js. It is a library that provides a set of features, like the possibility to handle requests with different HTTP methods (which on contrary must be programmed on your own by using Node.js), some basic web application settings like setting the port for connection or serving files.

To install libraries (packages), I will use the Node package manager tool (in short npm). With this software, it is possible to install all necessary packages from one place, so it is not necessary to access each package separately.

3.2 Frontend

Following section is dedicated to the frontend of the application. The frontend part of the application is written in React.

3.2.1 React

React [MP22] represents a Javascript library for more accessible user interface creation. It is currently managed by Meta Platforms which also developed Facebook. An advantage of React over other frameworks is the ability to divide the application's user interface into components. The component represents the fundamental element of the library. It typically represents a class or function. At the same time, it is possible to create bigger components from the smaller ones, and each can be used as separate, reusable parts.

The state of the application is saved by using the Redux package.

3.2.2 Redux

Redux [AtRda22] is a JavaScript library used to control the state of variables, i.e., what values variables acquire. It is used, for example, in combination with React. All data (or application status) processed by applications are grouped under one JavaScript object, called a store - Storage. As a result, it is possible to access all variables from one place, which is much clearer, and work with data is better manageable. Thus, Redux represents a model, a structure by which we control data access, change, addition, and deletion. Within Redux, we should not change the state of variables directly to avoid errors. Instead, actions are used to change values. An action is represented by a simple JavaScript object that describes what happened. The action must contain an attribute type. An attribute defines the type of action which is executed. Thanks to this, we know exactly what happened in the application and how it changed. In addition, we can define additional parameters to the action which serve as additional arguments (they are optional).

3.3 Backend

In this section, I will describe the backend part of the application before the changes, followed by an analysis and stages which were proceeded.

3.4 Frontend and Backend communication

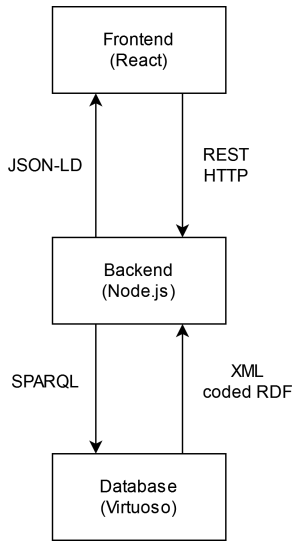


Figure 3.1: Communication between database, backend and frontend.

3.4.1 RTK Query

3.5 Functionality of existing application

Chapter 4

Analysis of issue

4.1 Existing issues

4.2 Required funcionality

Chapter 5

Design

This chapter is dedicated to the design of the new backend application. The current backend of the application will be replaced entirely with the new backend, as the former functionality is no longer needed after implementation. A different approach to data and its modification is used, with unified logic for the frontend part of the application.

5.1 Replacement of REST API to GraphQL

GraphQL uses a different approach to data as a REST API. When we fetch the data using the RESP API as a response, we will get the requested object with all its attributes except those representing another object. In this case, we must use join notation or send two requests (if we have two objects) to the server. This approach takes more computer resources and is more complex for a developer to implement. Also, we constantly receive the whole object and fields that we don't even need. This approach makes the responses through the network heavier. On the other hand, GRAPHQL uses a different approach and allows us to get in the response data that we need without

returning the whole object. Also, it supports getting the referenced objects as attributes without the need to do an explicit join or send two requests to the server. In this case, it is always treated as one request, and we get just one response. In the example, you can see how we can retrieve the user and their team using GRAPHQL and also the REST API.

```
GET /data/user/MnOaN&?_join=memberOf -- depth one
-- memberOf = TeamInstance
GET /data/team/X&aEnO -- depth two
--from TeamInstance we get Team identifier
--and then we can get the name of the Team
```

Listing 5.1: REST API call and join

```
query getInfoUserTeam{
  courses_User{
    courses_memberOf{ --type TeamInstace
      courses_approved --depth one
      courses_Team{
        courses_name --name of the team - depth two
      }
    }
  }
}
```

Listing 5.2: GraphQL abstract with depth data filtering

The additional advantage is that we can hide fields in the schema. If the field is not provided in the schema, it cannot be shown by GraphQL. With this, we can hide private data, like passwords, and not include them in the schema. To exclude them from the schema, we mark these fields as hidden,

and there will not be included.

5.2 Courses system schema

The schema of the Course system represents an ontology for this system. The schema was never fully described in RDF data, but in the backend part of the application, it is written in JS form (JS object). We need a complete RDF schema for UltraGraphQL to work with our DB. Otherwise, we are only able to query some of the required fields. Therefore, we create a script that must transfer the JS form into the RDF schema.

5.3 Script for creating a schema

The script must provide the complete RDF schema for our Course system ontology. In addition, each JS object must be mapped to RDF triples. For each JS object, there might be multiple triples provided. Therefore, a mapping must be made. Each model represents an instance of rdfs:Class which is saved in attribute type. The inheritance is allowed to be used; therefore, there exists an additional attribute subclassOf. This attribute references the parent model. Multiple subclasses are possible. Thus we can make multiple triples reference to the specific classes.

5.4 UPDATE of the existing fields

UltraGraphQL, by default, provides INSERT and DELETE actions. INSERT action is used to insert a new triple about existing or new IRI. In case we insert some attribute for the existing IRI, a new triple is made, and in the end, there are multiple queries with the same subject. In the sense of

data type, this is just adding a new element to an existing one. Therefore, its type is a collection (also called a list). When we call INSERT action, it will always add a new attribute without changing the existing one. The same also applies to the DELETE action. If we delete the identifier, it will delete all of its related triples. However, we can also delete just one specific triple, and if we have more triples with the same subject and predicate, it will just pop the specific one from the list. However, we would like to perform an action that will update the existing triple by replacing the value. Update action in SPARQL can be achieved by executing INSERT and DELETE queries. So there is no possibility of editing the existing one. However, if we INSERT and DELETE the query separately as two mutations, it would be run as two queries and two different transactions. We want to prevent that, and therefore an UPDATE action was created. Its purpose is to update the existing triple, with the logic of SPARQL, that the INSERT and DELETE script is called within one transaction. For that purpose, an additional action called UPDATE is added to the UltraGraphQL. Same as for the INSERT and DELETE actions, this action will also need access to the schema created for this action. After that, a specific query must be made that will perform delete and insert as one transaction - for the SPARQL, this is achieved by:

```
WITH <graphName> DELETE { ... } INSERT { ... } WHERE { ... }
```

Listing 5.3: UPDATE performed through SPARQL as one transaction

We will remove all triples where the subject and predicate are the same. It means that if there was an array of triples, the whole collection would be removed, and a new single value would be added.

Chapter 6

Implementation

In this chapter, I will describe the process of implementing the adjusments to UltraGraphQL and the refactoring which was proceed before the adjustments.

6.1 Create Script

In this section, I will describe the process of implementing the create script.

6.1.1 Design purpose

6.1.2 Implementation

6.2 Refactoring

6.2.1 Frontend issues fixes

6.3 UltraGraphQL adjustments

6.3.1 Initial state

6.3.2 Adjustments

6.3.3 New features

Chapter 7

Authorization

In this chapter, I will describe the process of implementing and structure the authorisation rules.

7.1 User authorization

7.2 Sensitive fields

Conclusion

Bibliography

[AtRda22] Dan Abramov and the Redux documentation authors. Redux. https://developer.mozilla.org/en-US/docs/Web/JavaScript/, 2022. Accessed: 2022-04-12.

[BGM14] Dan Brickley, R.V. Guha, and Brian McBride. Rdf schema 1.1. https://www.w3.org/TR/rdf-schema/, 2014. Accessed: 2022-07-12.

[BHBL11] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. In *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pages 205 – 227. IGI Global, 2011.

[CDG+14] Richard Cyganiak, DERI, NUI Galway, David Wood, 3 Round Stones, Markus Lanthaler, and Graz University of Technology. Rdf 1.1 concepts and abstract syntax. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, 2014. Accessed: 2022-04-02.

[Cif20] Milan Cifra. Semantic data model for a course management system. Master's thesis, Comenius university in BratislavaFMFI FMFI.KAI, Bratislava, July 2020.

[con22a] MDN contributors. Express/node introduction. `https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction/`, 2022. Accessed: 2022-20-11.

[con22b] MDN contributors. Javascript. `https://redux.js.org/`, 2022. Accessed: 2022-20-11.

[Fda21] The GraphQL Foundation and GraphQL documentation authors. Graphql. `https://graphql.org/`, 2021. Accessed: 2021-13-12.

[HKP+12] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. Owl 2 web ontology language primer. `https://www.w3.org/TR/owl2-primer/`, 2012. Accessed: 2022-08-12.

[MP22] Inc. Meta Platforms. React. `https://reactjs.org/`, 2022. Accessed: 2022-04-12.

[PS08] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf. `https://www.w3.org/TR/rdf-sparql-query/`, 2008. Accessed: 2022-07-12.

[SARB14] Guus Schreiber, VU University Amsterdam, Yves Raimond, and BBC. Rdf 1.1 primer. `https://www.w3.org/TR/rdf11-primer/`, 2014. Accessed: 2022-27-01.

[Sof20] OpenLink Software. Openlink virtuoso universal server documentation. `https://docs.openlinksw.com/virtuoso/`, 2020. Accessed: 2022-07-12.

[tc21] UltraGraphQL team and contributors. Ultragraphql. `https:`

//git.rwth-aachen.de/i5/ultragraphql/, 2021. Accessed: 2022-20-02.

[Wik21] the free encyclopedia. Wikipedia. Linked data. https://upload.wikimedia.org/wikipedia/commons/5/5d/Screenshot_from_2021-05-17_12-26-27.png, 2021. Accessed: 2022-04-12.

List of Figures