CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Attribute-Based Content Redaction in Large-Scale Data Systems

A Study of Granular Access Control

Master's thesis in Computer science and engineering

Jimmy Andersson
Claudio Aguilar Aguilar

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Attribute-Based Content Redaction in Large-Scale Data Systems

A Study of Granular Access Control

Jimmy Andersson
Claudio Aguilar Aguilar

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Attribute-Based Content Redaction in Large-Scale Data Systems
A Study of Granular Access Control
Jimmy Andersson
Claudio Aguilar Aguilar

Supervisor: Daniel Strüber, Department of Computer Science and Engineering
Advisor: Oscar Bäckström, Volvo Cars
Examiner: Gregory Gay, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Attribute-Based Content Redaction in Large-Scale Data Systems
A Study of Granular Access Control
Jimmy Andersson
Claudio Aguilar Aguilar
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Data privacy has become significantly more important over the past years, leading to new laws and regulations that citizens and organizations must abide by. As a consequence, keeping data from being exposed to the wrong audience is no longer just an interest of the individual - it is also a legal requirement on companies that collect and store sensitive information. Different geographical regions may also enforce different data privacy laws, making matters even more complex for organizations that operate on a global scale. On top of the regulatory aspects, internal information security policies may specify that some subsets of data must be shared differently depending on security classifications and who requests it. This master's thesis project conducts a Design Science Research study aiming to combine two existing techniques – attribute-based access control and redaction. The goal is to evaluate whether the resulting component is a viable approach to granular access control in request-response type APIs that expose sensitive data to a global audience. The study produces a Proof-of-Concept implementation as an artifact, which is evaluated and compared to the type of role-based RESTful APIs commonly used in industry today.

Keywords: Access Control, Large-Scale Data, Computer Science, Data Engineering, Redaction, Attribute-Based Access Control

Acknowledgements

We want to thank our academic supervisor Daniel Strüber for his advice and expertise, and our examiner Gregory Gay for constructive feedback and suggestions for improving this thesis. We would also like to thank our supervisor at Volvo Cars, Oscar Bäckström, for his guidance and support.

Last but not least, we want to extend our thanks to Elisabeth Prissberg, Jonathan Frisk, and Linnea Johnsson for all the rewarding discussions and support during this project.

Jimmy Andersson, Gothenburg, June 2022
Claudio Aguilar Aguilar, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1
Introduction

This chapter introduces a master's thesis intending to propose an attribute-based content redaction mechanism for large-scale data systems. The concept is developed in cooperation with the Global Online Experience division at Volvo Cars, which, for example, handles the online purchase experience for Volvo Cars customers.

1.1 Background

Data privacy has become significantly more important over the past years [1], leading to new laws and regulations that citizens and companies must abide by. As a consequence, keeping data from being exposed to the wrong audience is no longer just an interest of the individual - it is also a legal requirement on companies that collect and store sensitive information. On top of the legal aspects, companies also want to avoid the bad press that comes when unauthorized or unethical use of collected data is exposed to the public. One example of such an incident is the Facebook-Cambridge Analytica data scandal [2].

Collecting and storing data is something that many, if not all, companies and institutions do today. The data can range from information about employees, such as salaries, living addresses, and tax identification numbers, to information about products, sales, and production. Different data properties naturally come with different levels of confidentiality and sensitivity. For example, information about an employees living address might need restricted visibility, while some product information could be publicly accessible. On top of the confidentiality aspects, there is also the matter of following laws and regulations that dictate how and when different kinds of data could, and should, be made available.

Designing an automated software system to correctly evaluate when to share data, and with whom, is by no means a simple task. Such a design needs to take the four elements of privacy into consideration: *Purpose*, *Visibility*, *Granularity*, and *Retention* [3]. The *granularity* element dictates that a user should be granted access to any subset of information he or she is allowed to see. Together with legal requirements that differ depending on geographical region, this access evaluation can quickly become cumbersome and unwieldy to maintain.

1.2 Purpose

The goal of this project is to evaluate whether attribute-based redaction is a viable approach in large-scale data systems. The main idea is to provide highly granular data access while upholding strong privacy and confidentiality requirements. The project aims to propose a model for how to design such a redaction mechanism, as well as provide a Proof-of-Concept[1] (*PoC*) implementation for testing and evaluation.

The model describes the building blocks needed for an implementation of the proposed redaction mechanism, and states any assumptions or preconditions that must hold. It also describes the structure of a rule set that can be used to evaluate whether access is granted to a data property.

A PoC of the redaction mechanism is provided for multiple reasons. Firstly, to aid in testing, collection of measurements, and evaluation of the model. Secondly, to allow others to review and reproduce the results obtained in this project. The PoC will also include a mocked data set and a set of API implementations for testing and evaluating the redaction mechanism.

1.3 Research Questions

The main goal of this study is, as previously stated, to evaluate whether a combination of attribute-based access control and content redaction is viable in large-scale data systems. That is, however, too broad of a question for this project alone, and it is therefore boiled down to three more specific research questions.

- **RQ1**: Can an attribute-based redaction mechanism improve data granularity compared to common role-based RESTful approaches used in the industry today?
- **RQ2**: Is an attribute-based redaction mechanism as maintainable as common role-based approaches used in the industry?
- **RQ3**: Will an attribute-based redaction system experience a significant performance degradation compared to role-based RESTful approaches?

One important aspect with regards to *RQ1* is the prioritization of different attributes. For example, a company may want to withhold some subset of information if the request is made from a specific geographic region. However, they may also want some of their employees to have access to the aforementioned information even if they go on a business trip to said region. Looking at the attributes in the wrong order, or without taking interactions into account, could cause valid requests to be incorrectly denied.

[1] Available at github.com/JimmyMAndersson/attribute-based-content-redaction

The interactions between attributes mentioned above is also a potential source of combinatorial explosions, which may make a system difficult to maintain over time. *RQ2* serves to evaluate the potential of keeping rule sets and endpoints up-to-date, which is a critical aspect when choosing a model for a real-world project.

Given that combinations of a large number of attributes may be difficult and time-consuming to validate, one also needs to make sure that an implementation does not make the entire system feel sluggish or unresponsive. *RQ3* focuses on measuring whether an attribute-based redaction mechanism maintains the systems throughput, or whether it slows down the processing to unacceptable levels.

1.4 Scope and Limitations

The core of this project is to decrease the risk that an organization incorrectly shares information with parties that are not supposed to have it. A common case where such an event might happen is when the organization exposes some subset of data through a request-response type API. With this in mind, the project is scoped to study how a redaction mechanism can be applied to such an API.

The proposed solution will include a model of how to structure API endpoints, redaction components, and rule sets. It will also include assumptions and preconditions that need to be met in order for an implementation to work properly. Efforts will be put into making the PoC as generic as possible, to make it easier to generalize any findings and test them on other data sets.

While it is possible to implement granular redaction in RESTful APIs, their design principles have a few drawbacks that make it more difficult. Since data is separated through the use of many different endpoints, all of which may return very different response data structures, the redaction process becomes highly dependent on which endpoint it belongs to. As the response structures evolve over time, maintaining the redaction mechanism for each endpoint will likely become unwieldy and cumbersome. Instead, this project will work with a graph-based API design, which allows for very granular data access through a single endpoint. Graph-based APIs have become increasingly popular over the past couple of years, with almost 15% of developers using them in production [4].

Many APIs that are used in production offer options to mutate data, on top of the option to query it. However, since this project mainly addresses redaction of information that is supposed to be read, it will inherently put less effort into providing a way to mutate data. Mutation, and the possible expansion of the access control mechanism to cover such cases, will be included as a discussion point to aid further research.

Keeping information from falling into the wrong hands is a complex problem with many moving parts. This project will focus solely on using attribute-based redaction as a means of dynamically deciding which data to share, and with whom. It will

not put any efforts into other aspects of security that may also need to be addressed by practitioners. Such aspects include, but are not limited to, networking and IT infrastructure, injection vulnerabilities, and database introspection possibilities.

1.5 Thesis Outline

The remainder of this report will be presented using the following structure. Chapter 2 covers the theory on different access control models, redaction policies and API architectures, while chapter 3 talks about the research methodology applied for this project. Chapter 4 describes the proposed solution and presents the implementation of it. Chapter 5 illustrates the results collected after evaluation, and chapters 6 and 7 summarizes and concludes this thesis by discussing and reflecting on the outcome.

2
Theory

This section introduces various technical concepts that underpin the research made in this project.

2.1 Access Control Models

As the name suggests, access control refers to the concept of controlling access to some set of resources. The decision of granting or denying access to said resources can be informed by a number of different factors, which has given rise to a number of different access control models [5]. This section introduces two of the most commonly used ones.

2.1.1 Role-Based Access Control

In the 1970's, computer systems started featuring more applications and serving more users, which increased awareness of data security. It caused system administrators and software developers to focus on various kinds of access control models to ensure that only authorized users where given access to certain information [6].

The Role-Based Access Control (*RBAC*) model restricts access to resources based on the role of the requesting user. Roles are assigned to users according to their job title and position in the organisational hierarchy, and each role is granted permission to access relevant data.

RBAC is one of the most widely used access control models because of its ability to easily reassign roles as needed. However, it is also prone to a phenomenon called "role explosion", which appears in environments where granularity is important. For instance, the doctor-patient problem describes a scenario where RBAC becomes unwieldy [7]. Since patient records should only be shared with the treating doctor, each patient would give rise to a separate role in the system. On top of that, nurses and administrators may require access to partial patient records, further adding to the number of roles.

2.1.2 Attribute-Based Access Control

As the name implies, attribute-based access control (ABAC) admits access to some resource by evaluating whether some combination of attributes fulfill a given criteria [8]. The attributes may be associated with the requesting user, the requested

resource, or the environment in which the request is made. For example, users may only be allowed to see information about a new, unreleased car model if they work in a specific department and have the proper security clearance. This approach allows IT personnel to define fine-grained and complex access rules, making it well suited for today's complex IT environments.

One should be aware of, though, that ABAC is not a silver bullet. The attributes that are available to make decisions are arbitrary and may change at any point in time. This places higher demands on providing a management system that enables developers to keep attributes and rule sets up-to-date [9]. Attribute-based access control is also susceptible to "rule explosion", since adding an attribute can significantly increase the number of possible rule combinations.

2.2 Information Obscurement

When denying access to requested resources, one can do so in a number of different ways. Three relevant approaches are presented in this section.

2.2.1 Full Rejection

If any part of a request fails an access control check, the request is be denied in its entirety and an appropriate error is returned to the requesting party [10]. For example, a request could provide an invalid authorization token, and therefore be denied access and receive a 401 error. However, the request may also be denied because the response contains some subset of sensitive information, even if the requester is authorized to see some other subset of the response.

2.2.2 Full Redaction

If the requesting entity is allowed to see a subset of the response information, but is denied access to some other subset, the disallowed subset could be masked from the response [11]. In practice, such an operation could be performed in a variety of ways – for example by one of the following:

- The information is removed from the response structure, without any trace of it being there in the first place.
- The data keys are left inside the response, but their corresponding values are set to some placeholder values. Suitable placeholder values could be either the data type's "zero value", a value outside of the data domain, or the 'null' value.

The second approach entails the requesting parties being able to see that the denied subset of data exists, even if they are not allowed to access it. There are times when that knowledge could be helpful. However, it could also pose a data security risk under certain conditions.

2.2.3 Partial Redaction

Sometimes, individual data points can be classified as partially sensitive and partially public [11]. One such instance is a Swedish personal identity number, where the first six digits can be considered public, while the last four are sensitive. In such cases, one may choose to apply partial redaction by masking out the last four digits and replacing them with some placeholder value. For example, the Swedish personal identity number of a person born on January 1, 1990 could be partially redacted and shared as "900101-****".

2.3 API Architecture

The term "API architecture" refers to a set of high-level ideas for how to structure an interface which exposes data and functionality to external programs [12]. This section presents two approaches to design such interfaces; the RESTful approach, which is very common in the industry today, and the graph-based approach, which has recently started to gain traction.

2.3.1 RESTful APIs

REST was introduced in the year 2000 by Thomas Fielding and is one of the most common API protocol in the industry today. It relies on Uniform Resource Identifiers (URI's) to identify and interact with resources. Different operations are performed by transferring messages via common Hypertext Transfer Protocol (HTTP) methods such as POST, PUT, and GET [13].

A REST-based API makes data available through endpoints. Each endpoint returns data from a particular resource and each resource has a predefined set of fields [14]. Since each resource requires a separate request, API users often need to make several separate requests to collect information from multiple related resources [15].

2.3.2 GraphQL

In 2015, Facebook introduced GraphQL as a new way of structuring web APIs. Available data structures are defined using a schema, and are exposed through a single API endpoint [16]. Surveys conducted by RapidAPI in the past three years show that GraphQL is becoming more common, with the proportion of responding developers using it in production increasing from 6% to 14.7% [17, 18, 4]. Notable companies using GraphQL include Meta, KLM, Starbucks, and The New York Times, to name a few [19].

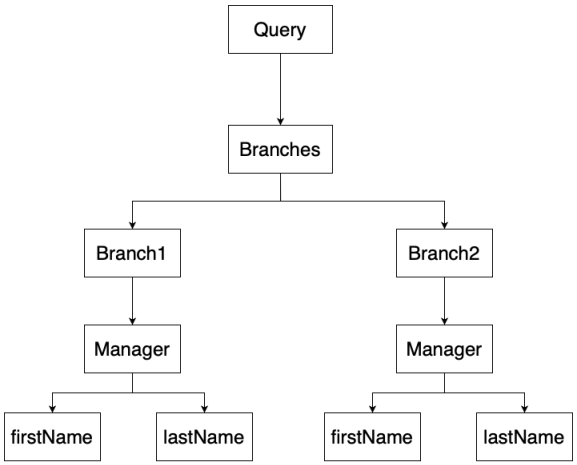
Contrary to RESTful APIs, GraphQL-based servers build their services around operations, instead of around resources. That is, instead of providing an endpoint per resource, GraphQL provides two different entry points inside its single endpoint - one for queries, and one for mutations. A Query type request fetches and returns

available information, while a Mutation type request has the ability modify stored data. In both cases, once the request has completed, a response is sent back to the user using JSON [16]. A more detailed description of the JSON format is available in the ECMA-404 JSON data interchange standard [20].

A GraphQL schema can be viewed as a multi-graph where each node is a data object [21]. Each node also holds a list of fields, which are defined by a name and a data type. When a field is requested, GraphQL maps it to real data by using a resolver function which specifies how to fetch the information. For example, the Query request shown in Figure 2.1a can be visualized using the tree structure in Figure 2.1b. Each node, starting from the Query node, uses a resolver function to map out an edge to another node. Once the execution has hit all requested leaf nodes, the mapped out graph is serialized and returned to the user.

```
query {
  branches {
    manager {
      firstName
      lastName
    }
  }
}
```

(a) An incoming query request.



(b) An execution flow diagram of the incoming query request.

Figure 2.1: An incoming query and its corresponding flow diagram.

2.4 Resource Access Evaluation

Applying access control to a resource naturally requires a specification of which preconditions need to hold for someone to be granted access. These preconditions come together to form a rule set, which together define how data is shared with both internal and external parties.

2.4.1 Boolean Expression Evaluation

Access control based on Boolean expression evaluation (*BEE*) is a concept introduced by D.V. Miller and R.W. Baldwin in 1989 [22]. It works by encapsulating every access decision into the following form:

- The sentence "<Subject> can <Verb> <Object>" is true or false.

The variables in the above expression can further be defined as:

- **Subject**
  The party asking permission to perform an action on a resource.
- **Verb**
  The action that *Subject* wants to perform.
- **Object**
  The resource on which *Subject* wants to perform the action.

For example, in a system that keeps track of employee information, the statement "Martin from HR can read the salary information of employees in the office" should be true. To arrive at that conclusion, a rule evaluation engine needs to match the Verb and Object to a Boolean expression which can be used to evaluate whether the Subject is given access. What that expression looks like is arbitrary and decided by the system administrators, but in the described scenario it is likely to take Martin's job title and department into account.

2.4.2 Common Expression Language

Common Expression Language (*CEL*) is a language specification created by Google [23]. It is designed to allow fast, type-safe, and memory-safe evaluation of expressions, which makes it an attractive choice for implementing access control mechanisms. It is also non-Turing complete by design, which decreases security risks involved with evaluating arbitrary expressions on a computer.

3
Research Methodology

This chapter describes the research method used to conduct this study. It also covers the process of designing and evaluating the PoC[1].

3.1 Research Method

Design Science Research (*DSR*) revolves around producing and refining an artifact that addresses a real problem, while also generating new knowledge to the scientific community. Given the practical nature of this project, this methodology was deemed an appropriate choice [24]. Firstly, the questions under investigation stems from real-world problems that industry struggles with today [25, 26]. Secondly, the matter of modelling, generalizing, and describing a solution that extends beyond the boundaries of an individual company provides value to academia as well [27].

The aim of this project is to produce an artifact that combines two existing techniques: attribute-based access control and redaction. The combination of the two techniques is a valid scientific contribution according to Staron [28].

3.2 Artifact Design Process

The artifact development process is split into iterative cycles as shown in Figure 3.1. All of the iterations include work related to each of the research questions. This is contrary to guideline 5 (G5: Shift Emphasis) given by Knauss [27], which suggests that each iteration should place more weight on one specific research question. The reason for this deviation is that the research questions in this project neither have a clear successive ordering, nor do they individually focus on the situation as it was before, during, and after the artifact came into being. Instead, the artifact is evolved based on feedback from stakeholders and improved knowledge about the problem domain, without placing restrictions on which cycle addresses which question.

[1] Available at github.com/JimmyMAndersson/attribute-based-content-redaction

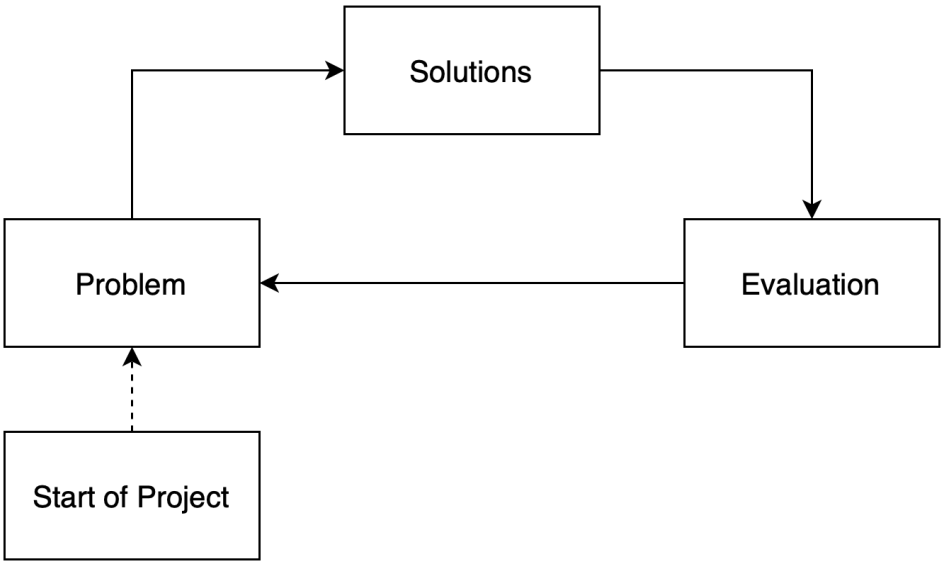


Figure 3.1: A visualization of the iterative nature of a Design Science Research project.

Each iteration consists of three stages, all of which are described below.

1. **Problem**
   The first stage of an iteration is focused on identifying and understanding one or more problems related to the research questions. This process is essential since it prevents the study from deviating from the scope [27], and is carried out in close cooperation with the stakeholders.

2. **Solution**
   Once the problems are sufficiently understood, the second stage of an iteration deals with designing and implementing a potential solution. The solutions may consist of constructs, models, and design theories derived from literature review and stakeholder discussions [24].

3. **Evaluation**
   The third stage of an iteration evaluates the effectiveness of the artifact based on its ability to provide a solution to the problems under investigation [24]. Both quantitative and qualitative measurements can be collected and used to evaluate the new solution.

3.3 Iteration 1

The first iteration of this project was naturally focused on the collection of design requirements and on crafting a first draft of an artifact that could be evaluated for the following iterations.

3.3.1 Problem

To get a deeper understanding of the problem, a series of unstructured discussion forums were held with stakeholders at Volvo Cars. These meetings served as a way to narrow in on the fundamental issues that needed to be solved, as well as their constraints. The requirements that came out of these meetings were as follows:

- The solution should allow users to receive the information they are granted access to, even if some subset of the information in the query is disallowed. Data that the user is denied access to should be redacted and replaced by a redaction value. For this implementation, the 'null' value was chosen as a replacement.
- The solution needs to be robust in the face of changes to the API. That is, if entry points are added or removed, the solution should require minimal maintenance efforts to support the new API surface.
- The solution should also be robust when the internals of an API change. If functionality is added, removed, or changed, the solution needs to be able to deal with those changes in a graceful manner.
- The solution needs to support a variety of data types - enough to cover the response types allowed by the ECMA-404 JSON standard [20].

Relevant literature and documentation regarding the available tool set was reviewed. The literature was found using keyword searches on IEEE Explore and ResearchGate. A meeting was also held with one of Volvo Cars' GraphQL experts. This further helped in understanding the problem set and possible directions to take when looking for a solution.

3.3.2 Solution

The first solution came down to designing an artifact that built access control into the default GraphQL data resolvers. By placing the decision of whether or not to share a property at the node level, the solution automatically adjusts and covers any request that can be put together by the user.

The solution also contained a first draft of a proposed structure and syntactic rules for a rule set that could be stored using a number of common data formats. The structure builds on specifying access rules for individual GraphQL schema type-property pairs, which allows practitioners to use local reasoning when defining rules. That is, decisions are made based only on the property and enclosing type a user is requesting, without regarding the path through the multi-graph one took to get there.

3.3.3 Evaluation

Evaluation for the first iteration was performed by collecting feedback from the stakeholders at Volvo Cars who were involved in the problem definition stage. The

feedback was gathered in semi-structured discussion forums, and concluded that the direction of the first solution was a compelling one. However, it also pointed out that lists of data suffered from a granularity issue. This point was brought into the problem formulation stage in iteration 2, and will be further discussed below.

The evaluation also contained small-scale, manual testing of correctness and performance. Those tests uncovered that while placing the access control in the default resolvers of GraphQL did yield the desired results for basic use cases, it failed in the presence of custom resolver functions.

3.4 Iteration 2

The second iteration put efforts into improving the general design of the redaction mechanism, trying to address issues and concerns that were raised or uncovered during the first iteration.

3.4.1 Problem

The second problem phase focused mainly on understanding two problems:

- The proposed solution worked well until introducing custom resolver functions. These bypass the default resolvers carrying the access control mechanism, and would therefore share any data that was requested. A user could potentially get a response containing a mix of values that either did or did not go through access control. The final solution would be required to catch both of these cases and route them through the redaction mechanism.
- Even though the first solution supported all the relevant data types, lists suffered from a granularity problem. Lists have a duality in that they can be viewed both as a single object, and as a composite of multiple objects. The first solution only considered the first alternative, and either shared the entire list, or nothing at all - leading to sacrificed granularity.

While the first issue impacted the general applicability of the solution, the second one had a more targeted negative impact on the research questions of this project. However, both of them needed to be addressed in order for the solution to be useful in real world scenarios and to address the overall problem statement.

3.4.2 Solution

To solve the issue with custom resolver functions evading the access control mechanism, the design needed to place the access control at a slightly different spot in the flow of execution. Instead of keeping it directly inside the resolver nodes, the mechanism was moved into wrapper objects instructed to encapsulate all resolvers - default and custom nodes alike. The new design guaranteed that all requested data would go through access control before being sent for serialization.

The list type granularity issue was resolved by adding a new set of rules to the already existing read rule. These new rules governed the behaviour of the access evaluation in the presence of list type properties, and caused it not only to consider the list as a whole, but also as a composite of individual elements. More details on the exact implementation of both of these solutions are covered in Chapter 4.

3.4.3 Evaluation

In the second evaluation stage, feedback was once again collected using semi-structured discussions with the Volvo Cars stakeholders that aided in the problem formulations. The conclusion was that the new solution seemed to fulfill all of the requirements that were set up in the first problem stage.

A more substantial set of automated performance tests were also conducted during the second evaluation. These, however, showed that the solution was painfully unable to compete with the type of role-based REST APIs that are commonly used in the industry today, as a request executing in 1 second on a RESTful API would take a minute and a half for the graph-based API to resolve.

3.5 Iteration 3

Since the second evaluation showed that adoption of this solution was not feasible from a performance point of view, the third iteration focused mainly on trying to make the solution computationally competitive.

3.5.1 Problem

The degree to which a new technology is useful is not only dependent on the assumption that it produces correct results and is easier to maintain than the alternatives, but it also requires the new solution to deliver similar performance characteristics. The second evaluation stage showed that this was clearly not the case with the proposed solution.

Some profiling showed that the main source of performance degradation was the dynamic compilation of Boolean expressions used to evaluate access in the access control component. The access control component was designed so that each node compiled all of the Boolean expressions it needed to evaluate access to its enclosed properties, even if another node had already compiled one or more of those rules before.

3.5.2 Solution

Solving the issue that prevented nodes from reusing access rule scripts required a redesign of how Boolean expressions were read, compiled, and passed between nodes in multi-threaded environments. The new solution implemented the following changes:

- Cache data structures were added to the rule evaluation component to avoid redundant compilation of access rules
- The rule evaluation component was redesigned to be thread safe, and a single instance could therefore be the sole authority for all access control nodes in a single request

3.6 Final Artifact Evaluation Setup

This section describes the design and technical aspects of tests used to collect measurements and data for analysis.

3.6.1 Hardware

All tests were run locally on a desktop machine in order to ensure that the same hardware profile was used for all of them. It also provided better control of other processes that were running on the same machine, compared to running in a cloud environment. The hardware specification for the test machine is listed in Table 3.1.

Model	Mac Studio
OS	macOS Monterey 12.3.1
CPU	Apple M1 Max 8 P-cores, 3.2 GHz 2 E-cores, 2.0 GHz
GPU	24 cores, 1296 MHz
RAM	32GB Unified Memory
Storage	512GB

Table 3.1: The hardware specification of the machine used to perform tests and data collection.

3.6.2 Evaluation Data Set

The data set used for evaluation consists of a mocked database resembling the company hierarchy of a big, international company. The decision to mock a data set was in part due to time management, and part due to the fact that we could design it to allow for many different kinds of queries. The latter would allow us to better test multiple kinds of requests and pinpoint bottlenecks associated with a specific type of query. The database files are available for inspection in the open-sourced PoC repository[1].

The database consists of two tables – one for all the employees of the company and another for the several corporate branches. Their respective fields can be seen in Table 3.2 and 3.3.

[1] Available at github.com/JimmyMAndersson/attribute-based-content-redaction

Field	Description
id	A unique identifier for an employee
first_name	An employee's first name
last_name	An employee's last name
title	An employee's job title
reports_to	An ID specifying an employee's closest boss
security_clearance	An integer defining the maximum confidentiality level of information the employee is allowed to access
branch	An ID for the office branch where an employee is stationed
salary	An integer specifying an employee's yearly salary

Table 3.2: Data set table fields for employees.

Field	Description
id	An unique identifier for an office branch
country	The branch's country of location
state	The branch's state of location
city	The branch's city of location

Table 3.3: Data set table fields for corporate branches.

3.6.3 Redaction Correctness

Before any other tests can be performed, it is crucial to make sure that the PoC works as intended and returns an accurate response. This task is performed using test cases, all of which contain four important pieces of information:

- The query for which to validate the response.
- The rule set the server should apply for the test.
- An authentication header that, if valid, provides the server with information about who is making the request.
- A JSON containing the expected response for the test case query.

A script is set up to automate the procedure of executing test cases and reporting deviations from the expected results. When launched, the script performs the following tasks:

1. Launch a GraphQL-based server employing the redaction mechanism.
2. Look up a test case that has not been run.
3. Tell the server to adhere to the test case rule set.
4. Send the test case query to the server and await its response.

5. Compare the servers response to the expected response and report the outcome.
6. The script starts over at 2 if there are more test cases to run, otherwise it shuts down the server and exits.

Test cases were designed to cover a wide range of scenarios, for example:

- Queries that form either wide or narrow tree structures.
- Queries that are either shallowly or deeply nested.
- Requests from different types of authenticated users, as well as from unauthenticated parties.
- Varying types of rule sets, especially with varying kinds of collection policies.

The expected responses were semi-manually assembled using SQLite JSON functions and operators. Using this functionality, SQL queries can be constructed that produce a JSON which matches the expected output of a particular query and rule set.

If the server response matches the SQL assembled expectation, the test case is regarded as being successful. It is, however, worth to note that the definition of a match is important in these tests. Since there are no guarantees that the server and the SQL query will assemble collection type properties using the same ordering, the matching algorithms need to take that into account and ignore the order of elements in such cases.

3.6.4 Request Execution Time

The performance aspect brought up by RQ3 focuses on the difference between the proposed solution and a common role-based RESTful API, and how fast they are able to resolve a request. Therefore, a rule set was set up that caused a graph-based server to mimic the response behaviour of a role-based REST server. The reasoning behind a setup like that was to make sure that both servers could produce the same responses to a set of requests, so that any measurable difference in execution time could be attributed to the process of resolving data.

The set of requests used to collect measurements were designed with a few important points in mind:

- The requests should be constructed so they allow both servers to produce identical responses with a single round trip request. Due to the general design principles of RESTful APIs, the data a user is interested in usually needs to be assembled from the responses of several requests. However, allowing such an approach would invalidate the comparison, since each REST response would be made up of multiple requests and their individual execution times.

- The requests need to cover a wide range of response sizes. That is, the responses need to contain a variety of resolved properties, so that it might be possible to draw conclusions about how the proposed solution behaves when resolving both small and large numbers of data points.

The measurement collection is performed using so-called "warm" servers, meaning that they have already received a number of requests before the collection of data starts. The reason for choosing this strategy is that both the Java Virtual Machine and database connections are able to make significant performance optimizations over time. Allowing the servers to receive 30 requests before measuring their performance was deemed sufficient to bring the execution times close to what they could look like over time. Once the servers had been warmed, 500 new requests were sent and their execution times recorded.

3.6.5 Survey

To evaluate the maintainability and the granularity's adequacy, the judgement of human expertise was necessary. For that reason, an online survey in the form of a questionnaire was chosen to collect qualitative data. This was done following the process and guidelines for conducting empirical studies in software engineering [29]. Due to the impracticality of collecting data from every single person that uses Volvo's APIs, a sample of the population were selected in the form of a non-probability sampling, in particular the technique "convenience sampling". This provided convenient people to act as respondents, such as experts regarding the topics provided in this project [29]. In order to get relevant and credible answers, the questionnaire was distributed through carefully selected internal channels at Volvo Cars.

The questionnaire adopted the following structure:

- **Problem statement**
  A formulation of the problem that the artifact is trying to solve.
- **Solution**
  A high-level explanation of how the artifact intends to solve the problem.
- **Knowledge & Experience**
  Questions regarding the respondents current knowledge and experience with GraphQL and access control models.
- **Introduction & Tutorial**
  A description of the artifacts functionalities. A short tutorial on GraphQL was also provided, since some knowledge about the technology was deemed necessary to answer the questions.
- **Maintainability & Usability**
  Questions to evaluate the maintainability and usability of the artifact. Using information about the configuration of a server and some response data, the respondents were tasked with deciding whether or not the response came from the suggested server.

- **Applicability**
  Questions regarding the perceived applicability of the proposed solution in the respondents own work. These questions also gave respondents the chance to provide notes about their experienced advantages and disadvantages of the model.

The responses were collected and analyzed in order to justify the research questions. The results can be observed in chapter 5, and the full questionnaire can be seen in Appendix A.1.

4

Implementation

This chapter describes the practical implementation of a GraphQL-based server with a builtin attribute-based redaction mechanism. Figure 4.1 shows an activity diagram that visualizes how access control is applied to a request, and can be used as a reference throughout this chapter.

4.1 Client-side

The client side consists of the interaction between a user and the server. This interaction boils down to two separate stages in the retrieval of data:

1. The user constructing a query and initiating a request to the server.

2. The server finalizing the request by returning a response to the user.

These stages are chronologically located at opposite sides of the data request, and the finalization stage does not contain much complexity. However, the initiation stage contains a few important steps that concern query construction and user authentication.

4.1.1 User Interaction

A user interacts with the server by constructing a query specifying which properties to retrieve. An example of such a query was shown in Figure 2.1a. The structure of the query allows the user to request arbitrary combinations of data points through a single endpoint. This makes for a simpler API surface when compared to a REST API, while still supporting very granular data requests.

The request should also contain some form of authentication token, if one exists, in order to provide the server with information regarding the requesting user. Given that a token is included, the server can construct a decision context containing user attributes to aid in sharing decisions.

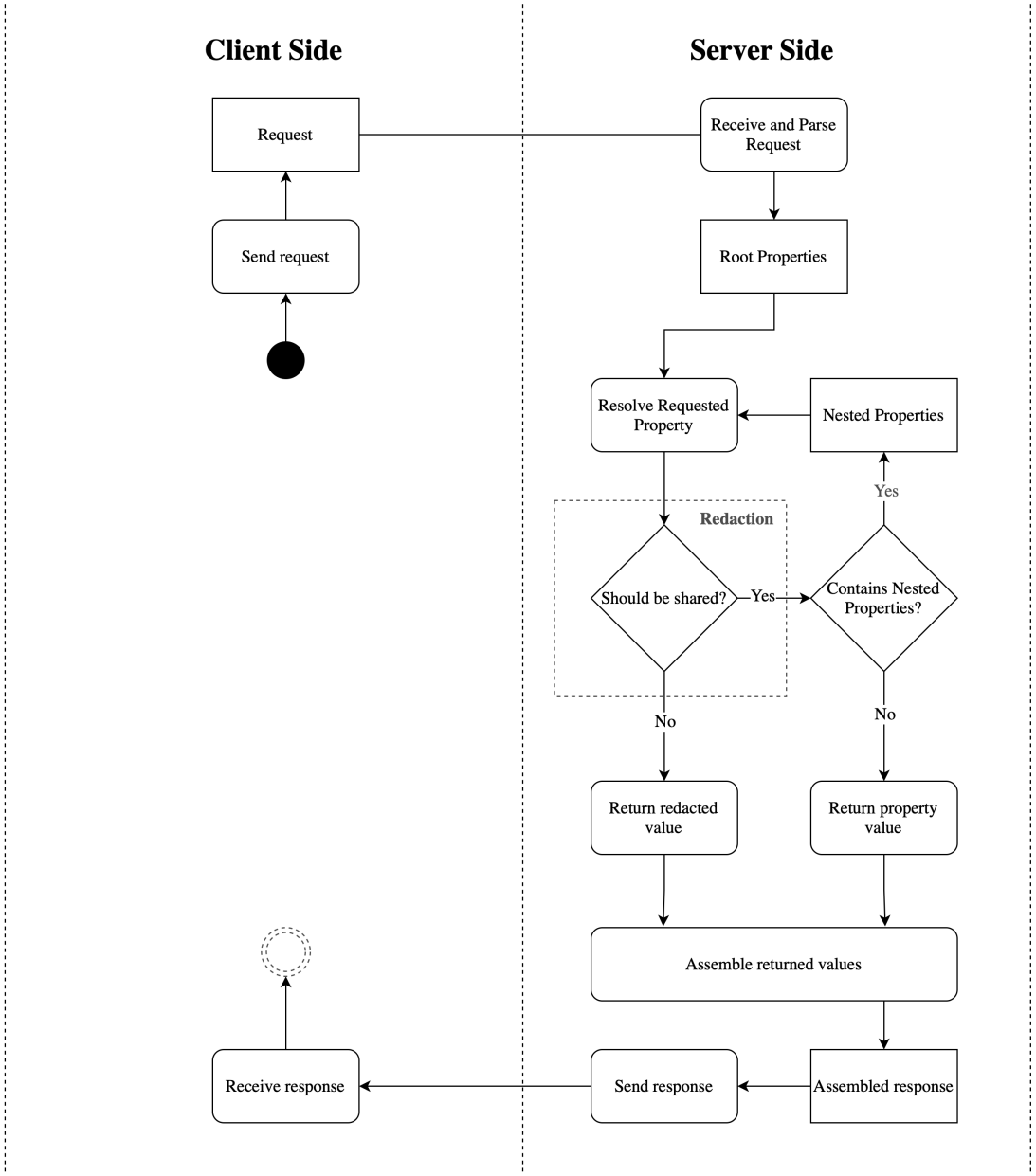


Figure 4.1: An activity diagram overview of the system.

4.2 Server-side

The server that handles incoming API requests must not only set up appropriate routing, but it must also supply the GraphQL instance with the information it needs to performs its tasks. This section describes the set up of a GraphQL server that supports attribute-based access control.

4.2.1 Receiving Queries

Whenever the server receives a request, it needs to set up a few structures in order for GraphQL to properly resolve the query.

1. **Subject Profile**
   For the Access Rule Evaluator to be able to function properly, it needs access to attributes describing the requesting party. This profile could, as previously mentioned, include authorization tokens, information about the Subject's employment or relation to the organization, and the Subject's security clearance level. The Access Rule Evaluator is covered in more detail below.

2. **Data Sources**
   When GraphQL resolves a query, it needs to fetch the data from somewhere. In practice, this could be as simple as including an in-memory data object. However, in large-scale data settings, it is more reasonable to believe that one includes database connectors or file handles.

3. **Access Rule Evaluator**
   When traversing the query graph, GraphQL will need to confer with an authority that knows which Objects the Subject is allowed to access. This component is discussed in greater detail in section 4.3.

The above structures are wrapped into a context object and added as execution input to the GraphQL instance. It is worth to note that the above list is by no means exhaustive of what one could or would like to include in an execution context. If the Access Rule Evaluator is programmed to make decisions on global attributes, one could include those in another structure as well. Such attributes could, for example, include the time when the request was received, the operating system being used, or any other attribute that is not specifically related to the Subject, but still may factor into the decision to grant access.

Apart from setting up an execution context, the server also needs to specify a global GraphQL directive. A global directive provides an injection point where one can execute arbitrary code before a property is resolved. It also provides access to the property's resolver function, which makes it a good place to intercept the resolved value and perform access control evaluation.

Figure 4.2 shows how the data flow is changed by inserting an access control wrapper between the GraphQL instance and the node resolving a data property. The wrapper acts as a resolver node in the eyes of GraphQL, but delegates the task of fetching data to the user-specified resolver function. When a value is returned, the access control node can perform value-level access control evaluation and return the appropriate final result to GraphQL.

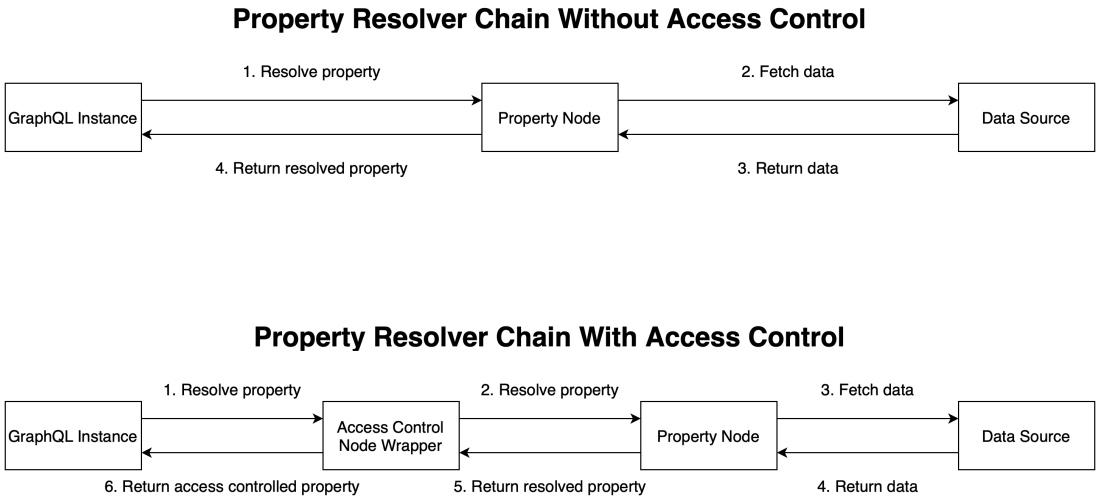


Figure 4.2: An illustration of how and where the access control mechanism intervenes when GraphQL resolves a data property.

An access control node's work can be divided into the following high-level steps.

1. Receive GraphQL's request to resolve a specific data property and pass it on to the corresponding resolver function.
2. Receive the resolved data from the resolver function.
3. Confer with the Access Rule Evaluator and return the resolved data or a redaction placeholder value, depending on whether the requesting party is authorized to access the information or not.

Once the property nodes have been resolved and all leaf nodes contain data points or redaction values, they can be serialized and sent back to the user.

4.3 Access Rule Evaluator

To properly evaluate whether or not a data point should be redacted, the nodes need to confer with an authority that knows how and when different information can be shared. In this system, the Access Rule Evaluator takes on the role of that authority. It uses an arbitrary number of provided attributes, such as authentication tokens, employment information, or the requesting party's relation to the organization, and provides all nodes with a single source of truth for sharing decisions throughout the entire request.

Since it is entirely possible that the system is distributed, it is also crucial that the Access Rule Evaluator delivers on some important requirements:

- **All instances need to act according to the same rule set**
  Obviously, even if an API service is replicated across different machines, both

the user and organization should be able to trust that all instances act according to the same rules. Therefore, it is important that potentially replicated Access Rule Evaluators stay in sync and react to rule set updates swiftly.

- **The rule set stays the same across a single API request**
  A server that receives several requests every second runs the risk of receiving a rule update in the middle of resolving a request. If that happens, the rules that were in place when the request started needs to persist until the response has been sent. If they don't, a requesting user could end up with a response that has been resolved using two different sets of rules.

The Access Rule Evaluator also needs to consider multiple cases when evaluating whether to include some resolved data in a response. These cases depend on the data types and are as follows:

- **The data is a singular property**
  In this context, a singular property refers to a data object that can be thought of as a single individual. Examples of common data types that satisfy this constraint are integers, floating point numbers, and text strings. Map and dictionary types also fit into this category, since they can be thought of as another branch node in the GraphQL query graph. For singular properties, the Access Rule Evaluator must answer the question "can <Subject> <Verb> <Object>?"

- **The data is a collection type property**
  A collection type property is allowed to contain zero or more data objects. Common data types that fulfill this criteria are arrays, lists, and sets, to name a few. This case is more complex than the previous, since it also requires the Access Rule Evaluator to decide whether or not the requesting party can have access to each of the elements inside the collection.

To cover all of these cases, the Access Rule Evaluator must be designed to take these differences into consideration. If a property is singular, a single rule can be applied to decide whether it should be shared. However, in the second case, the evaluator needs one rule to decide whether to share the collection, and another rule that decides which elements in the collection can be included.

4.4 Rule Set

To correctly grant or deny access to specific properties of the requested data, the Access Rule Evaluator must be able to make decisions on a very granular level. This requirement places high demands on the structure of the rule set, since a potentially large number of rules must be easily accessible by machines and easily readable by humans.

As shown in Figure 4.3, the rule set is structured according to a tree hierarchy with three available levels below the root node.

- **Level 1: Types**
  The first level specifies a type for which rules can be applied. One may, for example, want to impose some constraints on who can read employee information, and therefore specify an Employee node in the first level of the rule set tree.
- **Level 2: Properties**
  The second level specifies a property for which one wants to define a rule. An employee may have salary information that should only be available to certain individuals or departments, so one could insert a Salary node into the rule set tree.
- **Level 3: Rule Definitions**
  The rules used by the Access Rule Evaluator to determine access reside at the third level of the rule set hierarchy. The example in Figure 4.3 shows how different properties may have a different set of rule definitions available to them depending on whether they are singular or collection type properties. Three different categories of rules have been identified for the scope of this project: read, collection policy, and element filtering predicates. These are explained in more detail below.

Because the Access Rule Evaluator needs to handle different types of properties in different ways, the rule set needs to support that behaviour as well. Therefore, it allows up to three different rule categories to be defined:

1. **Read**
   The read rule is evaluated every time a resolver accesses a property, regardless of whether it is a singular type or a collection type. It contains a Boolean expression that evaluates to true if the requesting party is allowed to see the information.
2. **Collection Policy**
   The collection policy only applies to collection type properties, and defines how a list of objects is shared with the requesting party. It contains one of the following enumeration values:
   - **Full** - The collection is shared as-is.
   - **Redacted** - The collection is shared in its entirety, but with redaction placeholder values inserted at indices where the requesting party is not allowed access.
   - **Partial** - The collection is filtered so that it only contains data that the requesting party is allowed to see. The length of this type of collection is always less than or equal to the other two alternatives.
3. **Element Filtering Predicate**
   The element filtering predicate is a Boolean expression that evaluates whether an individual element in a collection type property should be shared. If this expression evaluates to false, the data is redacted according to the policy specified in the *Collection Policy* field.

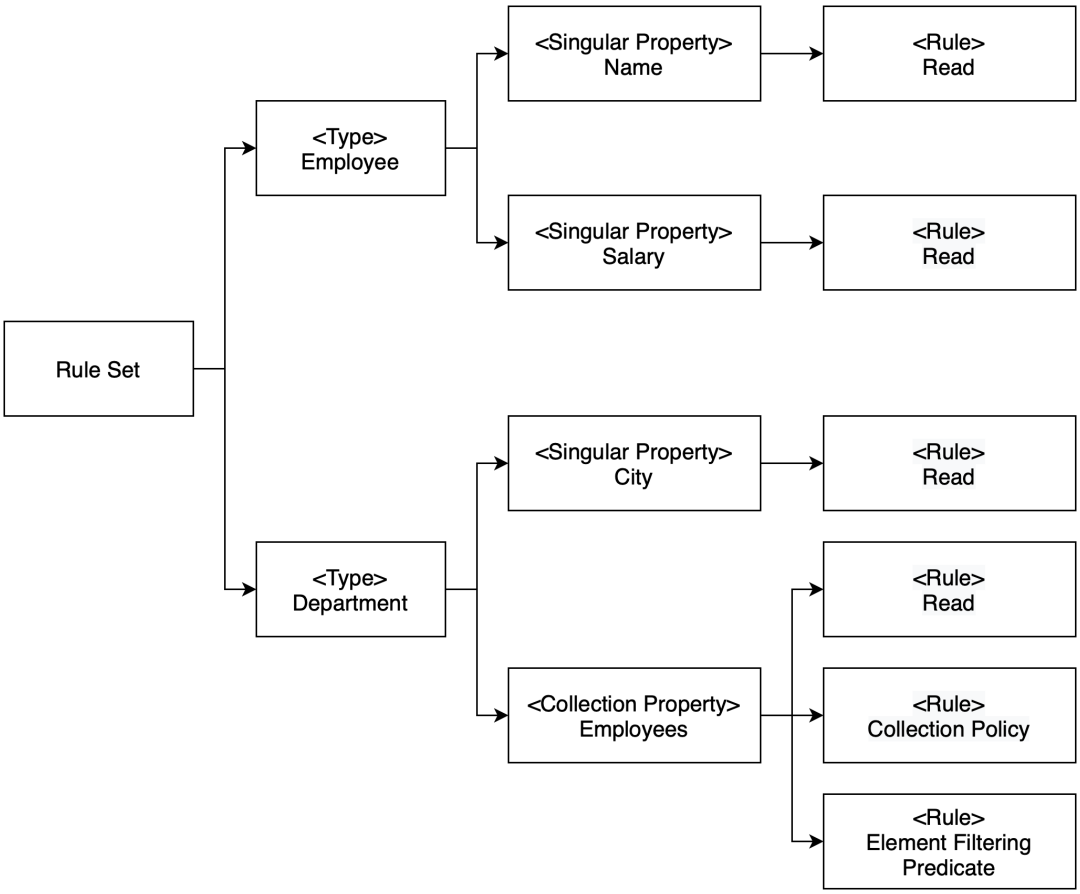


Figure 4.3: An example showing the general structure of a rule set.

The PoC[1] implements this rule tree structure using a simple JSON file. However, it should be noted that these rules could be stored using a number of different formats that support the expression of tree type hierarchies, for example YAML, XML, and relational database tables.

[1] Available at github.com/JimmyMAndersson/attribute-based-content-redaction

5
Evaluation Results

This chapter presents the collected measurements and the analysis of them. It starts by presenting the results from the correctness tests used to validate whether the solution works as expected, followed by a section on measurements regarding the performance of the system. Finally, it looks at the data collected from the survey.

5.1 Redaction Correctness

Table 5.1 shows the results from each of the redaction correctness tests. These measurements were collected as part of a set of automated tests, which were covered in section 3.6.3. A total of 14,377 access control decisions were made by the Access Rule Evaluator, all of which matched the expected outcome for the specified rule sets and queries.

Test	Access Control Decisions	Correct Decisions
1	2093	2093
2	1238	1238
3	1238	1238
4	4228	4228
5	1117	1117
6	1158	1158
7	3221	3221
8	82	82
9	2	2
Total	14377	14377

Table 5.1: Result summary for the redaction correctness tests.

5.2 Request Execution Time

The following results were collected as part of a set of automated tests, in a role-based REST server and a graph-based server were hit with a number of requests yielding similar responses. The full details of the setup can be found in section 3.6.4.

Figure 5.1 shows the average measured execution time per request, based on how many properties that request has to resolve. The plot shows that both the graph-

based and the REST-based solution tends to scale linearly with the request size.

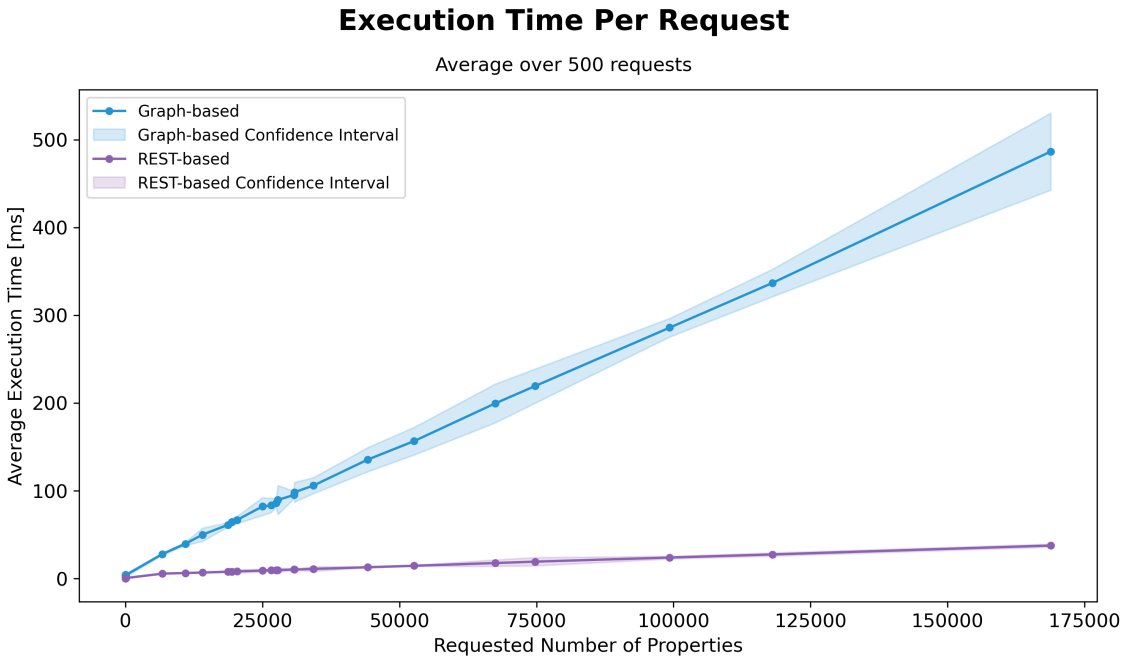


Figure 5.1: The average execution time for a request based on the number of requested properties. The plot also includes the 99% confidence intervals.

Figure 5.2 shows the calculated execution time per resolved property at different request sizes. One can see that the average time to resolve a property is larger for small requests, but approaches a steady state above a certain request size. The same pattern goes for both the graph-based and the REST-based solution, although the difference and variance for smaller requests is greater in the graph-based case.

The difference between the mean execution times is shown in Figure 5.3. This plot illustrates that the average execution time difference per property is large for smaller request sizes, but approaches a steady state of approximately 2.8 µs as the request size grows.

The calculated average execution time per property, as well as some statistical measures and computations, are presented in Table 5.2. The p-values come from a two-sample t-test, under the null hypothesis that there is no discernible difference in average execution time between the two servers. As can be seen, the p-values are very small and indicate significance across the board. One can also see that the difference in average execution time per property tend towards a steady state as the number of requested properties grows larger, as Figure 5.3 visualizes.

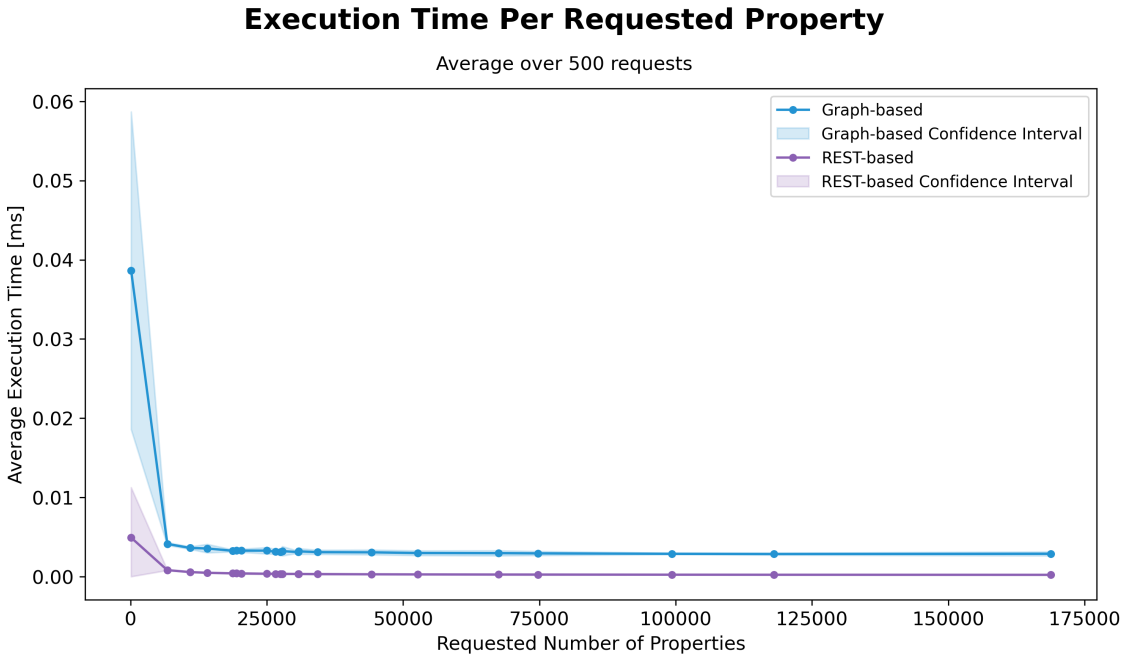


Figure 5.2: The average execution time to resolve a single property, based on the number of properties contained by a request. The plot also includes the 99% confidence intervals.

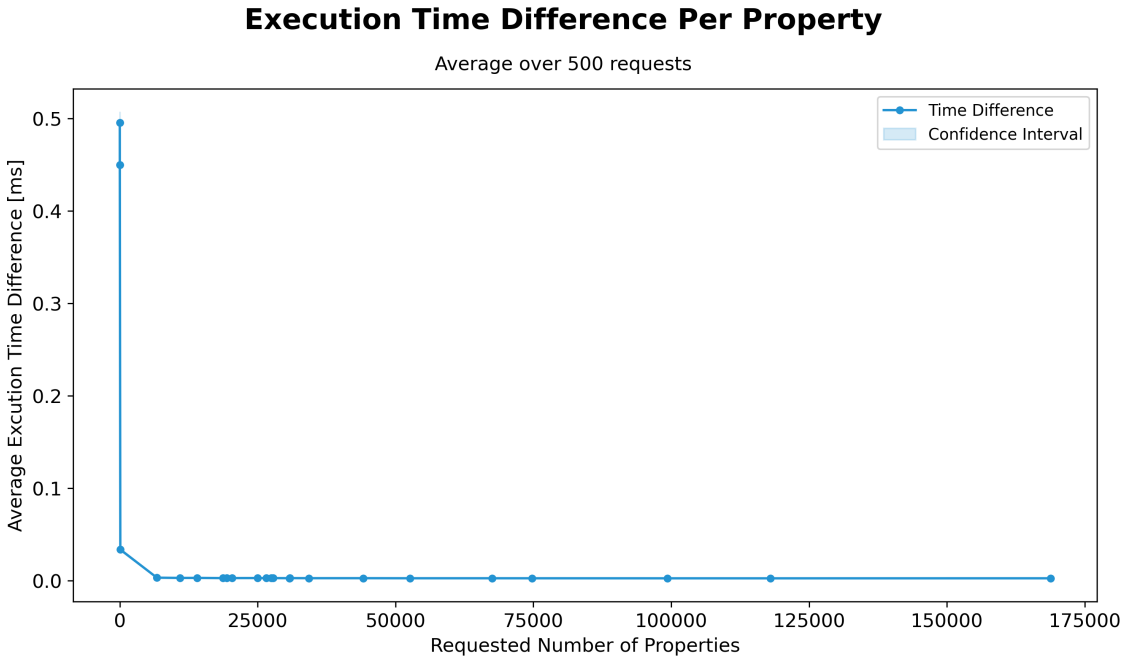


Figure 5.3: The average execution time difference between a request made to a role-based REST API and a graph-based server employing attribute-based redaction. The plot also includes the 99% confidence interval.

		REST		Graph-Based	
Nr of Properties	**P-Value**	**Average Time Per Property [s]**	**Standard Deviation**	**Average Time Per Property [s]**	**Standard Deviation**
5	< 0.0001	$7.773548e^{-}05$	0.000405	0.000573	0.000279
9	< 0.0001	$1.642575e^{-}05$	0.000043	0.000466	0.000089
101	< 0.0001	$4.918816e^{-}06$	0.000247	0.000039	0.000783
6734	< 0.0001	$8.159516e^{-}07$	0.000075	0.000004	0.000476
10937	< 0.0001	$5.696203e^{-}07$	0.000089	0.000004	0.000689
14060	< 0.0001	$4.809605e^{-}07$	0.000098	0.000004	0.002932
18713	< 0.0001	$4.112979e^{-}07$	0.000370	0.000003	0.000978
19424	< 0.0001	$4.098400e^{-}07$	0.001066	0.000003	0.001770
20342	< 0.0001	$3.980153e^{-}07$	0.000683	0.000003	0.001625
25004	< 0.0001	$3.597643e^{-}07$	0.000646	0.000003	0.003913
26588	< 0.0001	$3.527843e^{-}07$	0.000540	0.000003	0.003139
27506	< 0.0001	$3.456157e^{-}07$	0.000557	0.000003	0.001798
27821	< 0.0001	$3.435790e^{-}07$	0.000876	0.000003	0.006390
30755	< 0.0001	$3.332443e^{-}07$	0.000393	0.000003	0.001634
30836	< 0.0001	$3.291582e^{-}07$	0.000211	0.000003	0.004332
34283	< 0.0001	$3.174563e^{-}07$	0.001043	0.000003	0.003518
44192	< 0.0001	$2.896645e^{-}07$	0.000221	0.000003	0.005300
52661	< 0.0001	$2.750019e^{-}07$	0.000227	0.000003	0.006097
67502	< 0.0001	$2.604613e^{-}07$	0.001432	0.000003	0.008549
74756	< 0.0001	$2.563125e^{-}07$	0.001842	0.000003	0.007525
99299	< 0.0001	$2.401390e^{-}07$	0.000589	0.000003	0.004088
118019	< 0.0001	$2.320855e^{-}07$	0.000721	0.000003	0.006050
168815	< 0.0001	$2.223702e^{-}07$	0.000678	0.000003	0.016980
918208	< 0.0001	$2.012160e^{-}07$	0.002954	0.000003	0.010212

Table 5.2: Results from measuring execution time over different request sizes, and the p-values from performing a two-sample t-test under the null hypothesis that there is no difference between the two approaches.

The effect size was calculated using the Vargha Delaney A12 measure [30], and the result indicates a large effect.

$$A_{12} = 0.888889$$

5.3 Survey

The following results present the data collected from the survey. The collection was performed through an online questionnaire, as described in section 3.6.5. A total of 6 respondents participated in the survey.

In the first couple of questions, the respondents were presented with an access control configuration, as well as with a server request and some response data. Their task was to decide whether the request could have yielded the response, given the access control rules. Figure 5.4 shows a Likert plot describing the amount of effort respondents felt they had to put in to derive answers to these questions. As can be seen, none of the responses indicate that a lot of effort was needed. Summing up correct and incorrect answers, also shows that a total of 22 questions were answered correctly by the respondents, while only 2 were answered incorrectly.

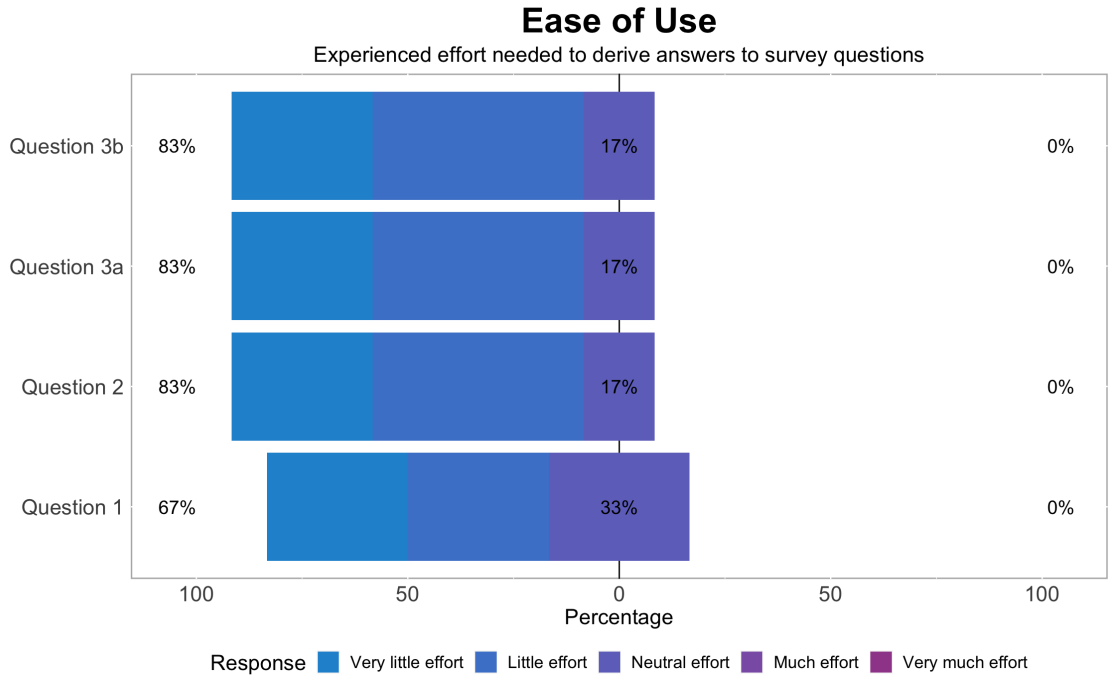


Figure 5.4: A Likert plot illustrating the amount of effort the respondents felt they had to put in to derive answers.

Figure 5.5 shows a histogram of how useful the respondents thought the solution would be if they had access to it in their own work. Looking at the plot, the respondents tend to rate the proposed solution either as not so useful, or as quite useful.

The free-form textual feedback collected in connection to this rating provides some insight into the perceived pros and cons of the suggested approach.

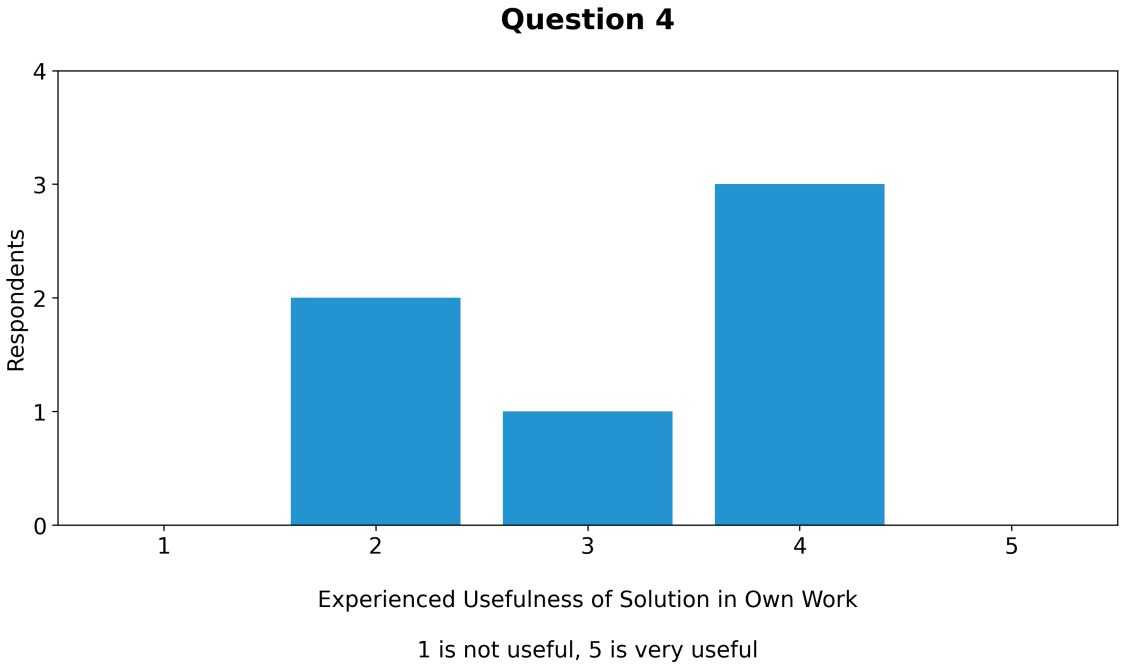


Figure 5.5: The distribution of how useful respondents thought the proposed solution would be in their own work.

The main advantages of the proposed solution circle around the rule set format. It is described as being a "pleasant format", as being a "compelling pattern", and as being a natural way to work with "whitelisting" data. Many of the respondents seem to agree that the rule set structure feels familiar and easy to reason about. The respondents also comment on the collection policy as being a useful component in the rule set.

On the cons side, some concerns are raised about the solutions ability to deal with complex rules. The concerns are not with regards to the structure of the rule set per se, but rather focus on the ability and feasibility to put together a decision context and user profile to support the access rule evaluator. As an example, "Person A should have access to Resource B for as long as he or she is assigned to Project C, and Project C is active" was brought up to illustrate that decision contexts may need to include a lot of information. All of this information would need to be loaded into memory at the beginning of a request, something that may be time consuming and cumbersome.

Another critique of the solution concerned the forced "nullability" in the API. The respondents raised concerns about the fact that they would have to redefine big parts of their public interface in order to satisfy this constraint. For the solution to be used, it would need to "support both schema-first and code-first implementation".

6
Discussion

This chapter summarizes the key results of this project and, in the context of those results, discusses the advantages and limitations of the proposed solution. It also makes recommendations for future research projects in related areas.

6.1 Redaction Correctness

The correctness tests cover different kinds of requests, rule sets, and requesting users, and the results show that all sharing decisions in these cases are made correctly. These results suggest that the mechanism will resolve a wide number of requests correctly, and that it can be considered accurate for a good portion of potential queries.

However, the tests do not cover every possible request, and may therefore miss edge cases where the implementation fails. Such cases could, for example, include requests with extreme query complexity, such as very deeply nested queries. Bugs, logical errors, and failures in edge cases are hard to avoid completely in any software, and their existence need to be considered even if these correctness tests are successful.

6.2 Performance

As could be seen in Figure 5.1, the graph-based solution is slower to resolve a request compared to a REST-based implementation. The performance impact is linear with respect to the number of requested properties. This impact can also be seen in Figures 5.2 and 5.3, which show the average execution time per requested property and the average execution time difference between the two implementations.

The *Common Expression Language* specification says that the language evaluates in linear time, which means that the number of variables evaluated to grant or deny access will also play a role in how much slower the graph-based solution is. The absolute time each variable adds to the execution was not measured or taken into consideration in these tests. Since the goal was to replicate the behaviour of a role-based REST server, only a single Boolean variable was used in each property rule during evaluation. This constraint needs to be taken into account when interpreting these results.

Looking at Figure 5.2 and 5.3, it is visible that the time difference per property between the two implementations is asymptotic. The graph-based solution is almost 8 times slower than the REST-based implementation for small requests, approaching an approximate 2.8 µs difference per property as the requested number of properties grows larger. This behavior can be attributed to the compilation of evaluation scripts, which takes place the first time when a specific property is resolved during a request. As the number of requested properties grows larger, the likelihood of evaluation scripts getting reused also grows and the compilation cost per property decreases.

A two-sample t-test shows that the execution time difference between the two implementations is well beyond statistically significant, even causing computers to round the p-value to 0. Furthermore, computing the Vargha Delaney A12 measure yields 0.888889, indicating a large effect size.

One thing to note in regards to the performance results above, is that the general design principles of RESTful APIs rarely allow a user to fetch all relevant data with one round trip request. Instead, one often needs to perform several network requests and assemble the desired information on the client-side. GraphQL, on the other hand, often serves all information readily assembled in one single request. This negatively impacts the overall performance of RESTful APIs for slightly more complex requests, and lessens the impact of the shown execution time difference. However, such scenarios were not measured because it would invalidate the direct comparison between the two approaches.

The decision to run all tests on a local machine also affects the performance test results. The graph-based server utilizes a large degree of parallelism, which means that the available hardware profile directly impacts the execution time. Cloud service providers usually allow their customers to pick a hardware profile for their environment, where more powerful profiles incur higher monetary costs. This relation implies, somewhat simplified, that if an organization spends more money on their cloud environment they will also cut execution times with the graph-based API. In the end, this makes the proposed solution susceptible to a trade-off discussion between performance and monetary cost.

6.3 Usefulness in Practice

Apart from the performance aspect, the perceived usefulness from a human point of view is also important to consider. One component of the usefulness is the ease of use from a human point of view. Figure 5.4 shows that the survey respondents felt like they needed to put relatively small efforts into figuring out the answers. The fact that the respondents got 22 out of 24 answers correct, in combination with the respondents difficulty assessments, suggests that the proposed solution is relatively easy to understand.

The survey also revealed another concern regarding the forced "nullability" of data. In order to use the proposed solution, developers may need to remove guarantees that a specific property always contains a value. This restriction goes against some design API principles, in which it is preferred to guarantee the existence of a value instead of sending optional data. To mitigate this concern, it would be fairly easy to extend the rule set structure to also support default values in case of redaction. Developers could, for example, choose whether a redacted value should be replaced by a "null" value, the data types "zero" value, or a custom value of the developers choice.

6.4 Threats to Validity

The cooperation with Volvo Cars offers many advantages, such as access to expert knowledge and industry experience. However, it also raises the question of how well the results of this study will generalize outside of the company [31]. Each organization is likely to have differing sets of IT infrastructure components, human resources, and organizational hierarchies – making each of them unique in their own way. Because of this, a well-suited solution for one company may not work for others.

One aspect of this concern is that the systems and data structures at Volvo Cars are unique to their use case and situation. Any solution tailored to their specific needs may end up being difficult, or even impossible, to use in other organizations. To mitigate this, the solution has been designed as a generic plug-and-play component, which can be easily injected into any existing GraphQL API server. This requirement enforces that the component is flexible enough to work with any information that API developers deem necessary for sharing decisions. By placing the definition of a decision context outside of the redaction component, the risk of tailoring a solution to one specific company decreases.

A second aspect regards which criteria a company should fulfill in order to find this solution useful. Volvo Cars, being a global automotive manufacturer with a clear software profile, have very specific use cases for these technologies. However, any organization that needs to expose potentially sensitive information to the Internet may benefit from the results of this study. Regardless of whether the use case concerns sharing company confidential data to other countries, or preserving GDPR compliance inside country borders, the results of this project can provide tools and insights for system architects and developers.

It should also be noted that the survey only had 6 respondents, which is indeed a small sample size. This fact needs to be taken into account when interpreting the qualitative results. Nevertheless, the respondents provided high quality textual feedback on the pros and cons of the solution, which supports a "quality over quantity" type of argument for including the survey results in this report.

The mocked data can also raise concerns regarding the generalizability of these results. In the real world, databases may suffer from various issues regarding data

normalization and information being scattered across a large number of tables in multiple databases. The fact that the PoC only uses a single database with two tables implies that the data is likely to be much cleaner and easier to access than in a real-world scenario. The main rationale for designing the mocked data like this was to remove as much undesired execution time variability and hidden complexity as possible, while still being able to test the solution properly. However, it is something that needs to be considered when interpreting the results.

6.5 Future Research

To make further progress in this topic, future research could focus on two problematic areas within the proposed solution.

- **Forced Nullability**
  The qualitative evaluation uncovered that the forced "nullability" in the current model may be undesirable from an API design point of view. Rather than using the "null" value by default, future research could investigate how to extend the current model to support arbitrary redaction values. These could possibly be incorporated into the rule set, but more efforts would need to be put into making sure that the solution does not become difficult to understand or work with.
- **Mutations**
  Another possible extension of the current model includes support for control of mutations. This project only investigated the outwards sharing of information, but a production server may also need to expose options to change and update data. Future research could look at the possibility of extending the current solution to cover mutability as well.

6.6 Answering the Research Questions

RQ1 asks whether the proposed solution can improve data granularity compared to common role-based RESTful alternatives. For this question, the answer must be that it can. The proposed solution does not deny access to an entire response simply because one or a few properties are denied, but instead masks those data points and sends the remaining information.

The answer to *RQ2* is not a straight forward yes or no. The proposed solution is definitely more maintainable when it comes to implementing and maintaining the redaction mechanism across endpoints and entry points, as all requests are resolved using the same data processing pipeline. This means that, compared to RESTful alternatives, there is no need to update multiple endpoint implementations if the rules or data structures were to change. The survey respondents also agreed that the rule format was easy to understand and work with. However, keeping the rule set up to date and synchronized with the properties available in the data structures

and decision contexts will add to the maintenance efforts, and the need to develop automated tools to aid in this task is evident.

Just as with *RQ2*, the results do not provide a clear cut answer to *RQ3*. There is indeed a significant difference in the execution time between the proposed solution and a common REST implementation, as shown by the hypothesis test. However, the performance degradation is not so big that it automatically disqualifies the solution from being used in a production setting. How big of a performance hit an organization is willing to take to get the upsides of the proposed solution is ultimately a call for the stakeholders. Nevertheless, the results suggest that the proposed solution can still be an attractive option for APIs where requests resolve up to a few hundred thousand properties. Above that, the added latency may become an issue.

7
Conclusion

This master's thesis has depicted the development of an access control mechanism building on two previously existing methods: attribute-based access control and redaction. The main goal was to produce a model for how to combine the two techniques and incorporate them into a request-response type server, as well as to answer some questions regarding the solutions maintainability, granularity, and performance characteristics. The proposed solution would have to offer a greater degree of granularity than commonly used role-based REST APIs. However, it also had to be easy to understand and maintain, as well as offer acceptable runtime performance.

The model uses a graph-based API architecture equipped with a "middleware" type of wrapper, that essentially intercepts and filters information on its way from the data source to the GraphQL instance. The filtering is performed dynamically by a central authority, which relies on a tree-like rule set structure to evaluate access conditions. The serialized response ends up containing only the data that the requesting entity is allowed to see, with all denied information being replaced by a redaction placeholder.

The model is deemed useful by the authors, as the results show that it correctly redacts information to which the requester is not granted access. It also incurs an acceptable performance hit for many use cases, since the added latency won't be truly noticeable for requests resolving less than a few hundred thousand properties. There are, however, problematic areas that need to be addressed within this model before it can be deployed into production. One concerns the forced "nullability" and the way that it affects the design choices of API designers. Another regards the ability to load and construct complex decision contexts to aid the Access Rule Evaluator in making access decisions.

Bibliography

[1] I. Oliver, "Experiences in the development and usage of a privacy requirements framework," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*, 2016, pp. 293–302.

[2] C. Criddle, "Facebook sued over cambridge analytica data scandal," BBC, London, United Kingdom, 2020, Accessed on: 2022-05-20. [Online]. Available: https://www.bbc.com/news/technology-54722362

[3] K. Barker, M. Askari, M. Banerjee, K. Ghazinour, B. Mackas, M. Majedi, S. Pun, and A. Williams, "A data privacy taxonomy," in *Proceedings of the 26th British National Conference on Databases: Dataspace: The Final Frontier*, ser. BNCOD 26. Springer-Verlag, 2009, p. 42–54. [Online]. Available: https://doi.org/10.1007/978-3-642-02843-4_7

[4] RapidAPI, "State of apis developer survey 2021 report," San Francisco, California, 2021, Accessed on: 2022-04-20. [Online]. Available: https://rapidapi.com/wp-content/uploads/2021/12/2021-Developer-Survey_Report.pdf

[5] S. Gentry, "Access control: Models and methods in the cissp exam [updated 2022]," Infosec Institute, Madison, Wisconsin, 2022, Accessed on: 2022-05-20. [Online]. Available: https://resources.infosecinstitute.com/certification/access-control-models-and-methods/

[6] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[7] X. Jin, R. Sandhu, and R. Krishnan, "Rabac: Role-centric attribute-based access control," in *Computer Network Security*, I. Kotenko and V. Skormin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 84–96.

[8] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, "Attribute-based access control," *Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[9] Chung, D. Ferraiolo, D. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (ABAC) definition and considerations," 2019-02-25 2019. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=927500

[10] M. Corporation, "Http response status codes," San Francisco, California, 2022, Accessed on: 2022-04-22. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[11] O. Corporation, "Database advanced security administrator's guide," Redwood Shores, California, Accessed on: 2022-04-22. [Online]. Available: https://docs.oracle.com/cd/E11882_01/network.112/e40393/redaction.htm

[12] RapidAPI, "Api architecture: Components and best practices for building robust apis," San Francisco, California, 2021, Accessed on: 2022-05-20. [Online]. Available: https://rapidapi.com/blog/api-architecture/

[13] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public rest web service apis," *IEEE Transactions on Services Computing*, vol. 14, no. 4, pp. 957–970, 2021.

[14] G. Brito and M. T. Valente, "Rest vs graphql: A controlled experiment," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 81–91.

[15] S. L. Vadlamani, B. Emdon, J. Arts, and O. Baysal, "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)*, 2021, pp. 10–17.

[16] L. Zetterlund, D. Tiwari, M. Monperrus, and B. Baudry, "Harvesting production graphql queries to detect schema faults," *CoRR*, vol. abs/2112.08267, 2021. [Online]. Available: https://arxiv.org/abs/2112.08267

[17] RapidAPI, "Rapidapi developer survey and insights 2019 - 2020," San Francisco, California, 2019, Accessed on: 2022-04-20. [Online]. Available: https://rapidapi.com/wp-content/uploads/2020/10/2020DevSurvey-Report.pdf

[18] ——, "Developer survey 2020 report," San Francisco, California, 2020, Accessed on: 2022-04-20. [Online]. Available: https://rapidapi.com/wp-content/uploads/2021/01/2020-2021-Developer-Survey-Report.pdf

[19] G. Foundation, "Who's using graphql?" San Francisco, California, 2022, Accessed on: 2022-06-05. [Online]. Available: https://graphql.org/users/

[20] E. International, "Ecma-404," Geneva, Switzerland, 2017, Accessed on: 2022-04-26. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

[21] G. Brito, T. Mombach, and M. T. Valente, "Migrating to graphql: A practical assessment," *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019. [Online]. Available: http://dx.doi.org/10.1109/SANER.2019.8667986

[22] D. Miller and R. Baldwin, "Access control by boolean expression evaluation," in *[1989 Proceedings] Fifth Annual Computer Security Applications Conference*, 1989, pp. 131–139.

[23] Google, "Common expression language – specification and binary representation," Mountain View, California, 2022, Accessed on: 2022-05-20. [Online]. Available: https://opensource.google.com/projects/cel

[24] J. v. Brocke, A. Hevner, and A. Maedche, *Introduction to Design Science Research*. Springer, Cham, 09 2020, pp. 1–13.

[25] T. Reuters, "Gdpr+1 year: Business struggles with data privacy regulations increasing," Eagan, Minnesota, 2019, Accessed on: 2022-05-20. [Online]. Available: https://images.thomsonreuters.com/Web/TRlegalUS/%7B04681875-adf4-4101-9a09-657385891ecf%7D_TR-GDPR-2019_v5.3final.pdf

[26] G. Lau, "Why companies are struggling with becoming gdpr compliant," London, England, 2021, Accessed on: 2022-05-20. [Online]. Available: https://www.business2community.com/strategy/why-companies-are-struggling-with-becoming-gdpr-compliant-02421083

[27] E. Knauss, *Constructive Master's Thesis Work in Industry: Guidelines for Applying Design Science Research*. IEEE Press, 2021, p. 110–121. [Online]. Available: https://doi.org/10.1109/ICSE-SEET52601.2021.00021

[28] M. Staron, *Action Research in Software Engineering: Theory and Applications*. Springer, Cham, 01 2020.

[29] T. Punter, M. Ciolkowski, B. Freimut, and I. John, "Conducting on-line surveys in software engineering," in *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, 2003, pp. 80–88.

[30] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000. [Online]. Available: https://doi.org/10.3102/10769986025002101

[31] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Transactions on Software Engineering and Methodology*, vol. 27, pp. 1–51, 07 2018.

A

Appendix 1

A.1 Survey Questionnaire

This questionnaire was sent out after the last iteration of this study. The set of questions are the following, where those that contain a '*' are regarded as mandatory.

Attribute-based Content Redaction

This form evaluates the design of a new access control mechanism for large-scale data systems. The project is a master's thesis made in cooperation between two Chalmers master's students and Volvo Cars.

* Required

Problem

Global organizations need to adhere to multiple laws and regulations, where different kinds of data may need to be shared in different ways depending on geographical region. For example, GDPR may treat some types of sensitive information differently than China's Cybersecurity Law, forcing an organization to share and withhold information in different ways.

The organizations own policies also dictate how to share data on top of the requirements made by law. This may include stricter stances on issues that are already addressed by law, or it may concern company confidential information. For example, a company might be okay with sharing salary information with members of the organization inside country borders, but may not want that information to be visible outside of said country. Another example might be that only members of a limited set of departments and countries should have access to blueprints of products that are under development.

Solution

The developed Proof-of-Concept handles access control by turning each attempt to read data into a question of the form: "Can <Requester> do <Action> on <Resource>?"

This question is asked for each and every requested data property, providing highly granular data access. In those cases where a user is not granted access to a specific piece of information, that data can be redacted from the response while still responding with the subset of information where access was granted.

To illustrate using another example, if Martin works in the Swedish HR department and wants to see the salaries of employees in Gothenburg, the rule evaluation engine might look at Martin's position in the company hierarchy, his security clearance, and Martin's geographical position to evaluate whether he should be granted access. In the response, Martin may find both real data and redaction placeholders, depending on whether he is granted access to a specific employees salary information.

Survey Overview

This survey takes approximately 15-20 minutes to complete. The general structure consists of:

- A few questions about your current experience with some of these topics.
- An introduction and small tutorial of the new system.
- Some questions regarding the maintainability and usability of the solution.
- Some questions about the applicability of this solution in your own work.

Your Current Experience

This section asks a few questions about your current experience and knowledge with some of the topics that are discussed in this project.

1. How much experience do you have working with GraphQL? *

Mark only one oval.

	1	2	3	4	5	
No experience	◯	◯	◯	◯	◯	Much experience

2. How much knowledge do you currently possess on different access control models such as Attribute-based Access Control or Role-based Access Control? *

Mark only one oval.

	1	2	3	4	5	
No knowledge	◯	◯	◯	◯	◯	Much knowledge

How It Works

This project studies a new access control concept for request-response API's exposed to the Internet. It does so by combining attribute-based access control and redaction. This section introduces how the mechanism grants access to different resources on a very granular level.

If you are already familiar with how GraphQL works, you can skip ahead to the Rule Set section.

Schema

The basic functionality of this API builds on GraphQL schemas. Entry points are specified by declaring properties inside of a special root type called "Query". As an example, Figure 1 shows a hypothetical entry point that fetches a random integer.

To request a random number from a server exposing this entry point, one sends an HTTP POST request with the following payload:

```
query {
  randomNumber
}
```

The response comes in form of a JSON string, which looks like the following:

```
{
  data: {
    "randomNumber": 42
  }
}
```

Figure 1

Custom Response Types

It is often the case that we want to specify more complex return types than a single integer number. GraphQL allows us to do that by specifying our own types. Figure 2 shows an example of an entry point that fetches information about a company employee by his or her ID number.

To see the first and last name of an employee with ID number 5, one would send the following request:

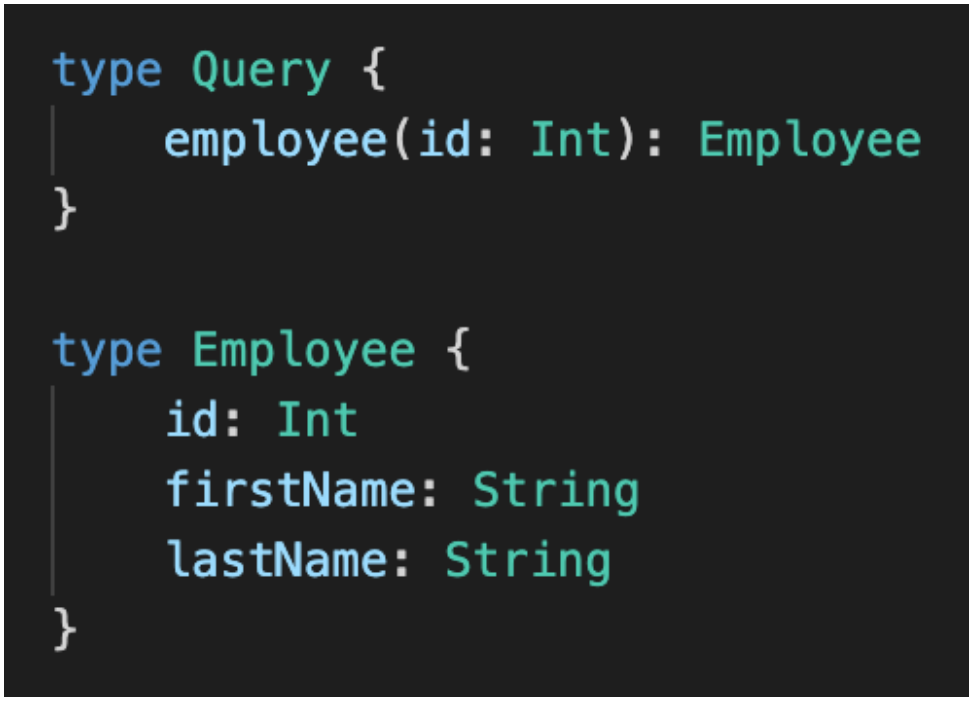
```
query {
  employee(id: 5) {
    firstName
    lastName
  }
}
```

Since our custom type is a composite of other properties, we need to specify which of those nested properties we would like to see. In this example, we only want the first and last name - not the id number.

The above query would generate the following response, containing only the properties we specified in our query:

```
{
  data: {
    employee: {
      "firstName": "John",
      "lastName": "Doe"
    }
  }
}
```

Figure 2



Rule Set

A rule set is the source of truth for which properties a requesting user is allowed to see. It consists of one or more Boolean expressions that need to evaluate to true in order for a property to be shared. If the expression evaluates to false, the property is redacted and replaced by a 'null' value.

If we would like to specify an access rule for the names of an employee, we could write a JSON rule set file that looks like the following:

```
{
  "Employee": {
    "firstName": {
      "read": "user.isAuthenticated"
    },
    "lastName": {
      "read": "user.isAuthenticated"
    }
  }
}
```

Now, if an unauthenticated user tried to perform the same employee information request as above, the response would look like the following:

```
{
  data: {
    employee: {
      "firstName": null,
      "lastName": null
    }
  }
}
```

Chained Rule Evaluations

The rule set allows us to define rules for both the Query type and for any custom types we may construct. As a consequence, one may need to fulfill multiple rules in order to get access to the desired property. For example, let's add a rule to the Query types 'employee' property:

```
{
   "Query": {
      "employee": {
         "read": "user.department == 'Human Resources'"
      }
   },
   "Employee": {
      "firstName": {
         "read": "user.isAuthenticated"
      },
      "lastName": {
         "read": "user.isAuthenticated"
      }
   }
}
```

Now, imagine that a user working in the 'Engineering' department makes the same request as in the 'Custom Response Types' section. Even though he or she is authenticated, they will still be denied access to see the information because of the rule that grants access to the 'employee' property. Instead, they will be presented with the following response:

```
{
   data: {
      employee: null
   }
}
```

Notice that the response cuts more information from this response, since the rule that first denied access was found earlier in the path towards the first and last name.

Available Information for Making Decisions

To make decisions, the rule evaluation engine needs access to relevant information. Developers are able to provide the engine with any amount of information they deem necessary to make appropriate access decisions. This information made available inside the rules through three different variable names:

user:
This variable contains information about the requesting user. In the earlier examples, it contained a property called .isAuthenticated, but it may also contain information about the users relation to the organization, their geographical location, their timezone, or any other information that is deemed appropriate.

object:
This variable is provided by the rule evaluation engine, and contains the object enclosing the property we want to evaluate access for. For example, when we decide whether or not to grant access to a firstName property, the object variable will contain the Employee that possesses that name.

value:
This variable contains the value we may grant access to. In the earlier example, the value variable would contain the String "John" when evaluating access to the firstName property of the Employee with ID 5.

Collection Properties

Lastly, we need to discuss the case where our API response includes a 'Collection Type'. This is merely a fancy name for a JSON list, and can be represented on the server side by a list, array, set, or a similar collection of elements.

When encountering a 'Collection Type' property, the rule evaluation engine needs to make multiple decisions:

1. Is the user allowed to read the list at all?
2. If they are allowed, which elements do they have access to?
3. If the user is allowed to see some of the elements, but not all, do we still send 'null' values to mark the redacted elements?

To support these decisions, 'Collection Type' properties support two more access rule types, apart from the 'read' rule:

collection_policy:
This rule specifies how the contents of a list are shared and supports one of three different values.

- full:
The list is shared as-is, in its entirety, without redacting any elements. This value can be used to indicate that the list does not contain any sensitive elements, and does not need any filtering. However, this does not exclude the possibility that the elements consist of nested properties that may be redacted at a later point in the process. For example, a list of employees may be shared in full, but the salary information of each employee may be redacted at a later point.

- redacted:
The full list is shared, but some elements may be replaced by 'null' values if the user is not allowed to see them. This option entails that the list may contain a mix of real data and redaction placeholders, all depending on whether the user is allowed access to individual list elements.

- partial:
The list is filtered, and elements that the user is denied to see are completely removed from the list; the user won't even know they exist. A partial list never contains any redaction values, but its length may be less than the original list if the user is denied access to some elements.

element_filter:
This rule contains a Boolean expression that decides which elements in a 'Collection Type' property should be redacted. The expression evaluates to true if the user is allowed access to the element. Because of how the collection_policy rule is defined, this evaluation only occurs when the collection_policy rule is either 'redacted' or 'partial'.

Collection Property Examples

Let's look at an example to fully grasp how all the rules interact for 'Collection Type' properties. Let us say that our API defines an endpoint to look up information about the two employees in our company, John and Jane. The schema could look like in Figure 3.

Let's start by defining a simple rule set that looks like:

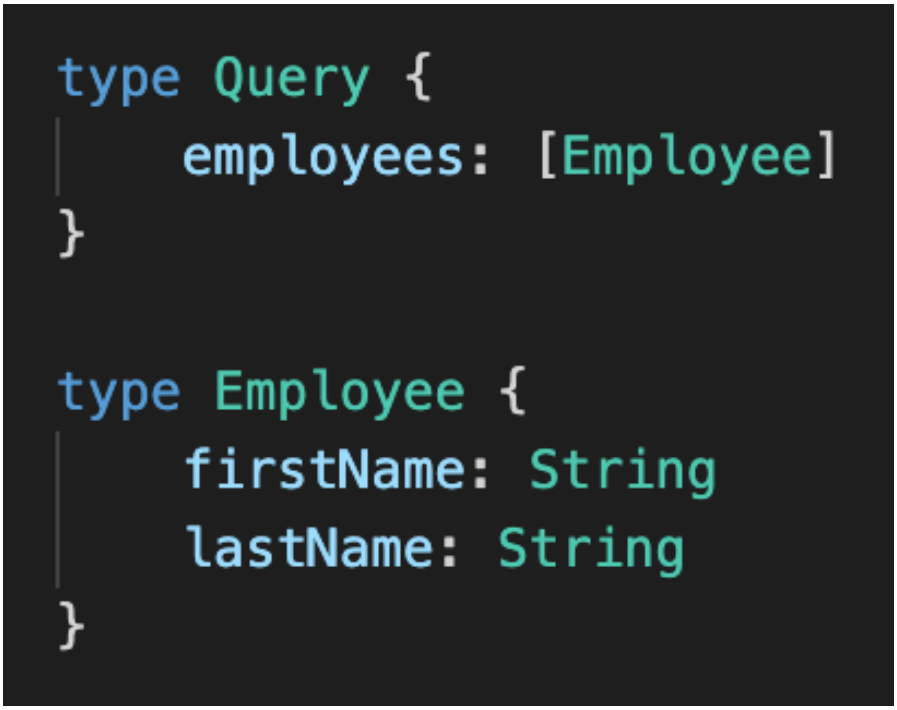
```
{
   "Query": {
      "employees": {
         "read": "true",
         "collection_policy": "full",
         "element_filter": "element.firstName == 'John'"
      }
   }
}
```

When the server evaluates a request, it first looks at the 'read' rule. Since it always evaluates to true, we will always be able to have access to the 'employees' list.

Secondly, the rule evaluation engine looks at the 'collection_policy' rule. Since we have defined that the employee collection can be shared as-is, the server knows that it is safe to return the entire list at this point. The user receives a response that looks like so:

```
{
   "data": {
      "employees": [
         {
            "firstName": "Jane",
            "lastName": "Doe"
         },
         {
            "firstName": "John",
            "lastName": "Doe"
         }
      ]
   }
}
```

Figure 3



If we were to change the 'collection_policy' to 'redacted', the rule evaluation engine would need to continue by matching each list element to the 'element_filter' rule. Since we are only allowed access to elements where the 'firstName' property is 'John', the algorithm will redact the other elements. The response looks like the following:


```
{
  "data": {
    "employees": [
      null,
      {
        "firstName": "John",
        "lastName": "Doe"
      }
    ]
  }
}
```


Likewise, changing the 'collection_policy' to 'partial' will filter the elements according to the 'element_filter' rule. The response would look like below:


```
{
  "data": {
    "employees": [
      {
        "firstName": "John",
        "lastName": "Doe"
      }
    ]
  }
}
```

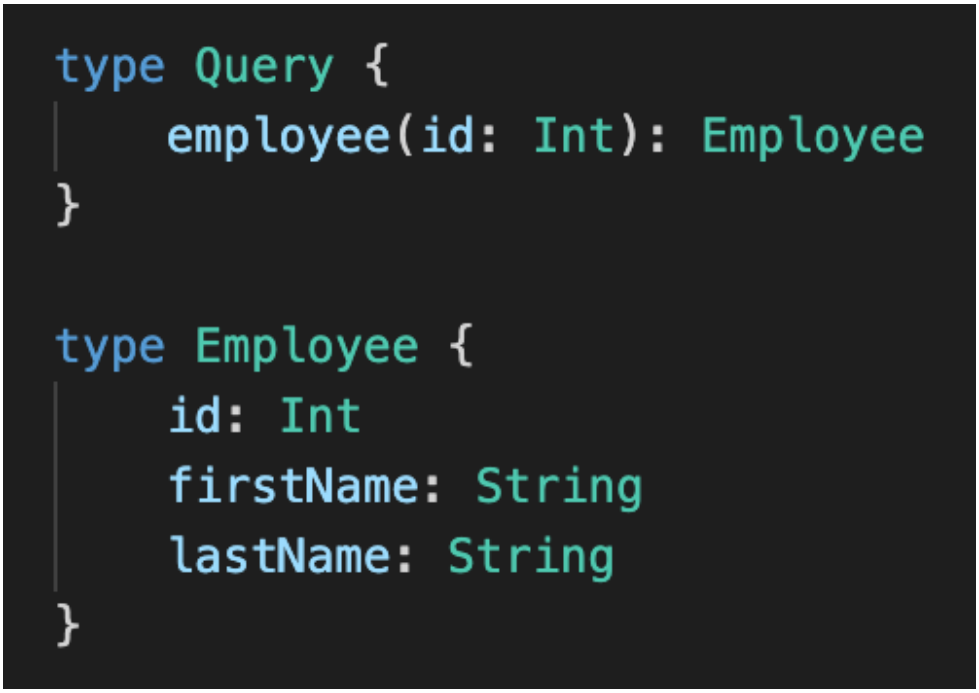
On to the Real Questions

Now you know the important bits of how the redaction mechanism works, and it has come time to ask you some questions to see where the design can be improved.

Question 1

The figures below show you a schema, a rule set and a response to use in this question.

Schema



```
type Query {
    employee(id: Int): Employee
}

type Employee {
    id: Int
    firstName: String
    lastName: String
}
```

Rule Set

```
{
    "Query": {
        "employee": {
            "read": "user.isAuthenticated"
        }
    },
    "Employee": {
        "id": {
            "read": "user.isAuthenticated && user.department == 'Human Resources'"
        },
        "firstName": {
            "read": "user.isAuthenticated"
        },
        "lastName": {
            "read": "user.isAuthenticated"
        }
    }
}
```

You are an authenticated user working in the 'Engineering' department. You request to see the information of the employee with ID number 2 by sending the following query:

```
query {
  employee(id: 2) {
    id,
    firstName,
    lastName
  }
}
```

3. Given the schema and rule set above, does this response give you the expected information? That is, does it give you access to all the information you are allowed to see, while redacting any information you are not allowed to see? *

Mark only one oval.

- ◯ Yes
- ◯ No

4. How much effort did it take to derive the answer to this question? *

Mark only one oval.

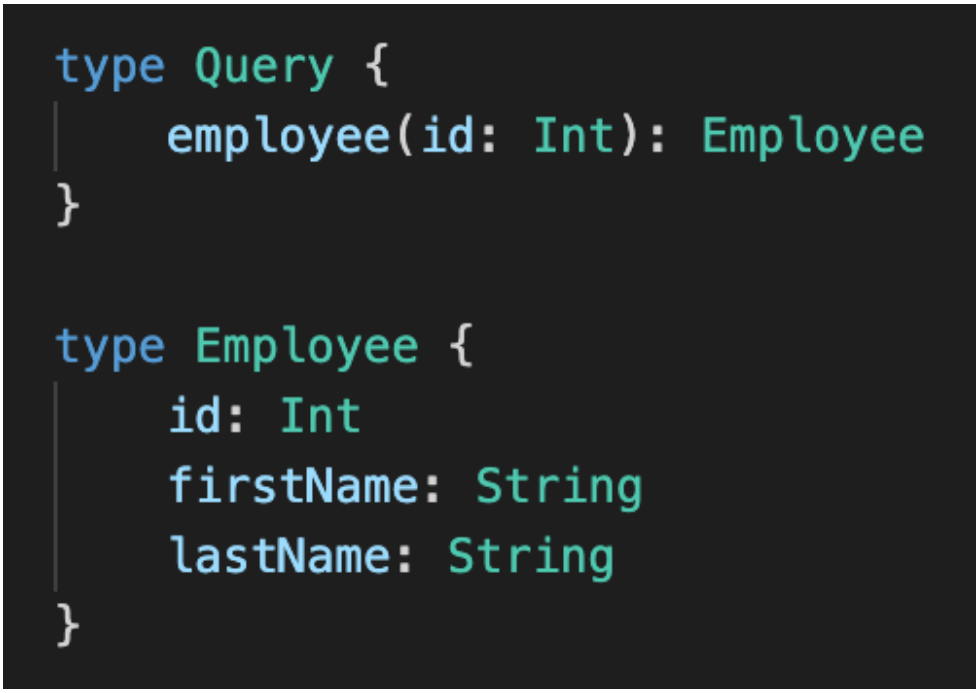
	1	2	3	4	5	
Very little effort	◯	◯	◯	◯	◯	Very much effort

5. Optional: Do you have any additional comments about this question?

Question 2

The figures below show you a schema, a rule set and a response to use in this question.

Schema



Rule Set

```
{
    "Query": {
        "employee": {
            "read": "user.isAuthenticated"
        }
    },
    "Employee": {
        "id": {
            "read": "object.firstName == 'John'"
        },
        "firstName": {
            "read": "user.isAuthenticated"
        },
        "lastName": {
            "read": "user.isAuthenticated"
        }
    }
}
```

Just like in question 1, you are an authenticated user working in the 'Engineering' department. You send another request to see the information of the employee with ID number 2:

```
query {
  employee(id: 2) {
    id,
    firstName,
    lastName
  }
}
```

6. Given the schema and rule set above, does this response give you the expected information? That is, does it give you access to all the information you are allowed to see, while redacting any information you are not allowed to see? *

Mark only one oval.

- ◯ Yes
- ◯ No

7. How much effort did it take to derive the answer to this question? *

Mark only one oval.

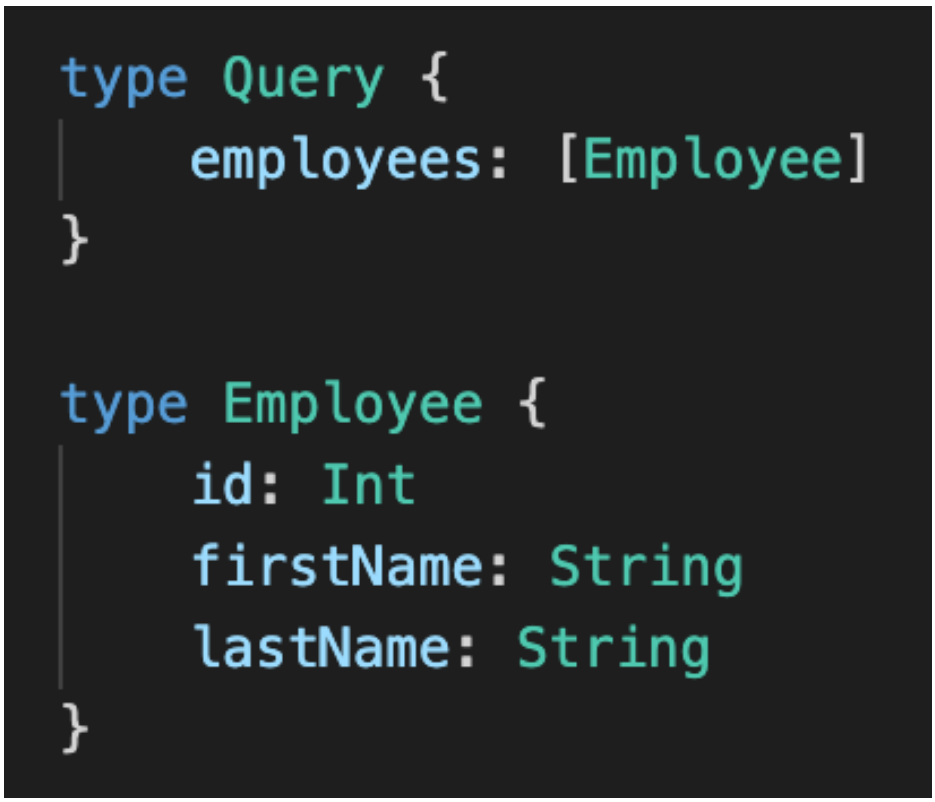
	1	2	3	4	5	
Very little effort	◯	◯	◯	◯	◯	Very much effort

8. Optional: Do you have any additional comments about this question?

Question 3

The figures below show you a schema, a rule set and a response to use in this question.

Schema



Rule Set

```
{
    "Query": {
        "employees": {
            "read": "true",
            "collection_policy": "redacted",
            "element_filter": "user.department == 'Human Resources'"
        }
    },
    "Employee": {
        "id": {
            "read": "user.isAuthenticated"
        },
        "firstName": {
            "read": "user.isAuthenticated"
        },
        "lastName": {
            "read": "user.isAuthenticated"
        }
    }
}
```

You are an authenticated user working in the 'Engineering' department. You send a request to fetch some information about all company employees:

```
query {
  employees {
    id,
    firstName,
    lastName
  }
}
```

9. Given the schema and rule set above, does this response give you the expected information? That is, does it give you access to all the information you are allowed to see, while redacting any information you are not allowed to see? *

Mark only one oval.

- ◯ Yes
- ◯ No

10. How much effort did it take to derive the answer to the above question? *

Mark only one oval.

	1	2	3	4	5	
Very little effort	◯	◯	◯	◯	◯	Very much effort

11. Which collection_policy rule would you specify if you wanted the above request to produce the following response? *

Mark only one oval.

- ◯ full
- ◯ redacted
- ◯ partial

12. How much effort did it take to derive the answer to the above question? *

Mark only one oval.

	1	2	3	4	5	
Very little effort	◯	◯	◯	◯	◯	Very much effort

13. Optional: Do you have any additional comments about these questions?

Question 4

This section provides a mix of multiple choice questions and free-form text fields, allowing you to provide more nuanced answers.

14. Based on your own experiences, how helpful do you think this solution would be in your own work? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	◯	◯	◯	◯	◯	Very helpful

15. Which benefits do you foresee if you had access to this solution?

16. What improvement suggestions do you have for this solution?