Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Secure infrastructure for exchanging rules in static code analysis tools

Timo Pohl

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Sara Ghatta
Commenced:	October 15, 2020
Completed:	April 15, 2021

Abstract

In software engineering, static code analysis can be used to inspect code and detect security vulnerabilities even in early stages of the development. This is done by analyzing a piece of code against a set of rules. The aim of this work was to create a secure data exchange infrastructure for static code analysis tools and providers of the rules being used. This enables these tools to update their set of rules by downloading the latest rules from rule providers. First of all, a research on alternatives for possible rule exchange infrastructures was done. During this, many existing data exchange and update protocols were examined. Then the requirements engineering and the search for technologies and protocols was conducted. Based on these results, the rule exchange infrastructure was designed. During the whole process, security was of utmost importance, but also requirements like maintainability and expandability were taken into account.

Contents

List of Figures

List of Listings

Acronyms

AEAD Authenticated Encryption with Associated Data. 37

AES Advanced Encryption Standard. 38

AOP Aspect-oriented Programming. 43

API Application Programming Interface. 27

BSI Federal Office for Information Security. 13

CRUD create, read, update and delete. 14

CSS Cascading Style Sheets. 45

DHE Diffie-Hellman key exchange. 38

DNS Domain Name System. 34

FAB Floating Action Button. 48

FTP File Transfer Protocol. 28

GCM Galois/Counter Mode. 38

GUI Graphical User Interface. 14

HAL Hypertext Application Language. 29

HATEOAS Hypermedia as the Engine of Application State. 28

HMAC Keyed-Hash Message Authentication Code. 39

HTML Hypertext Markup Language. 45

HTTP Hypertext Transfer Protocol. 28

HTTPS Hypertext Transfer Protocol Secure. 37

IDE Integrated Development Environment. 15

IETF Internet Engineering Task Force. 37

IoT Internet of Things. 13

IP Internet Protocol. 34

IT Information Technology. 13

JSON JavaScript Object Notation. 20

JWT JSON Web Token. 30

MVC Model View Controller. 45

MVVM Model View ViewModel. 45

NIST National Institute of Standards and Technology. 17

ORDBMS Object-Relational Database Management System. 33

POJO Plain Old Java Object. 41

RDBMS Relational Database Management System. 33

REST Representational State Transfer. 27

RPC Remote Procedure Call. 28

SAST Static Application Security Testing. 13

SHA-384 Secure Hash Algorithm 384. 38

SQL Structured Query Language. 33

TCP Transmission Control Protocol. 28

TLS Transport Layer Security. 17

UI User Interface. 45

URI Uniform Ressource Identifier. 28

URL Unique Ressource Locator. 29

UTC Universal Time Coordinated. 30

UUID Universally Unique Identifier. 29

VCS Version Control System. 15

XML Extensible Markup Language. 20

1 Introduction

1.1 Motivation

Internet security is getting more and more important, not least because of the drastically growing numbers of internet attacks. Companies all over the world are having huge amounts of monetary damage caused by cyber crime [ICR19]. Especially in times of the Internet of Things (IoT), there is a need to counteract this development and do whatever possible to ensure security in applications, especially in those connected to the internet.

A central part of cyber security is cryptology. As developing cryptographic procedures by oneself often goes wrong, they are and should be brought to applications using secure and well reviewed libraries. These build an essential basis for security in the Information Technology (IT). But often detailed knowledge on underlying cryptographic procedures is required [NKMB16], although one would expect that this is abstracted by the library. This problem can not be solved by simply updating and improving these libraries, because they always have to be compatible to existing applications. Since there may exist big security vulnerabilities, it is of utmost importance to detect and then repair a wrong usage of cryptographic libraries as early and as fast as possible.

This can be achieved by doing Static Application Security Testing (SAST), preferably already in early stages of development. Therefor SAST tools statically, which means without executing, evaluate a set of code syntactically against a set of rules. A problem with today's tools is that these rules are manufacturer-dependent. This means that it is not possible for a tool to analyze the code using all existing rules.

To solve this problem, a standardized rule exchange protocol as well as a rule provider infrastructure is needed. A SAST tool using this protocol and infrastructure would then be able to download all existing rules. These could be retrieved from official bodies like the Federal Office for Information Security (BSI) as well as from commercial IT security providers. When using the set of rules from several providers, it is possible to achieve a high vulnerability coverage and therefore it is nearly impossible to miss a security breach.

1.2 Goal

Currently, developers of SAST tools equip them with their own rules. Since there is no standardized rule syntax and no standardized, secure rule exchange protocol, rules created by other providers can not be used. In addition to that, experts can not simply create and release rules, they also have to release the analysis techniques.

The objective of this work is the creation of a secure rule exchange infrastructure, with which SAST tools can download and so update their set of rules from several providers. Also, rule providers should have a Graphical User Interface (GUI) to administer the rules they created. This means, they should be able to perform at least basic create, read, update and delete (CRUD) operations.

1.3 Task

In scope of this work, an optimal architecture for exchanging rules based on existing exchange infrastructures should be created. This architecture has to enable a SAST tool as client to download rules from an authorized provider. Therefor, the focus is on security. Especially the following features are required:

- Clients have to be sure the downloaded information originates from the source (authorized provider) they requested it from. This means, they must be able to detect whether an attacker tampered the data while they were sent from rule provider to client.
- Only authorized staff of a specific rule provider should be able to administer the rules created by this provider. Unauthorized staff as well as attackers should be unable to add, edit or delete rules.
- The infrastructure has to be maintainable and expandable. This, for example, allows both creators of SAST tools and rule providers to extend the basic infrastructure and add functionality.

Regarding the conception of the architecture, value is placed on platform independence on server side (rule provider) as well as on client side (SAST tool). Additionally, the server side should not rely on a specific web server. This is due to rule providers should not have to revise their existing infrastructure, what they probably would have to if a specific web server would be required.

1.4 Structure of this work

In chapter 2, the existing work of the university of stuttgart and RIGS IT GmbH regarding SAST is presented. After that, in chapter 3 some basics are given, including important security standards, the definition of data security and significant interface design rules. Following that, chapter 4 presents and explains the functional as well as non-functional requirements that together form the basis of the rule exchange infrastructure. Then, chapter 5 provides the actual specification and design decisions for each component of the rule exchange infrastructure. The penultimate chapter presents specific concepts and technologies that should be used when it comes to implementing the infrastructure. Finally, chapter 7 summarizes the results of this work and provides an outlook on the next steps.

2 Related work

Today, static code analysis can be used to detect vulnerabilities cheaper, faster and more reliable than manual reviews [JSMB13]. But there are also several problems with currently existing tools. A big problem is that they are not well integrated in the development process [SAE+18], as many tools are a standalone application and completely independent from the Integrated Development Environment (IDE).

In addition to that, the analysis often takes a lot of time. This is problematic, because it delays the time at which a vulnerability is detected and the later this detection takes place, the more complex it is to repair. For instance, it could have already made its way into the Version Control System (VCS) and thereby into other branches of the project. For this reason, from a developer's point of view it would be desirable to get a live feedback in the IDE, so that a vulnerability can be fixed right away. As there are many different IDEs, SAST tools and programming languages, a standardized integration of SAST tools into IDEs is not an option.

To address these problems, the university of stuttgart together with RIGS IT GmbH founded the criticalmate project. One of the main objectives of this project is the creation and implementation of a SAST-server as well as a uniform SAST-protocol. The criticalmate SAST-security-server is part of a customer's infrastructure and uses fast data flow analysis procedures to detect failures in the code. If it detects a wrong usage of a cryptography library, it will make use of the SAST-protocol to communicate with the IDE and inform it about this incorrect use. Based on this feedback, the IDE then can show a detailed error message to the developer, telling him what the failure is about and how it may be fixed. This process enables the developer to realize and fix security breaches right away. On the bottom line, this means any SAST tool which supports this SAST-protocol is able to communicate with any IDE which also supports the protocol. Of course this only works if the static code analysis techniques are fast enough to provide a live feedback and at the same time do still detect vulnerabilities reliably. For this reason, the development of fast and reliable data flow analysis techniques is also part of the criticalmate project.

Another important objective of the project is the rule exchange infrastructure, which is the focus of this work. Figure 2.1 shows the components involved in the criticalmate SAST and how they work together.

In the criticalmate project, the university of stuttgart in its sub-project focuses on the decoupling of rules and static code analysis tools. In principle, the following two things are important for this:

- a uniform, machine-readable rule notation and
- a general rule exchange infrastructure.

Additionally, the university of stuttgart wants to use artificial intelligence to automatically derive rules and improve the IDE messages shown to the developer [RWWM19].

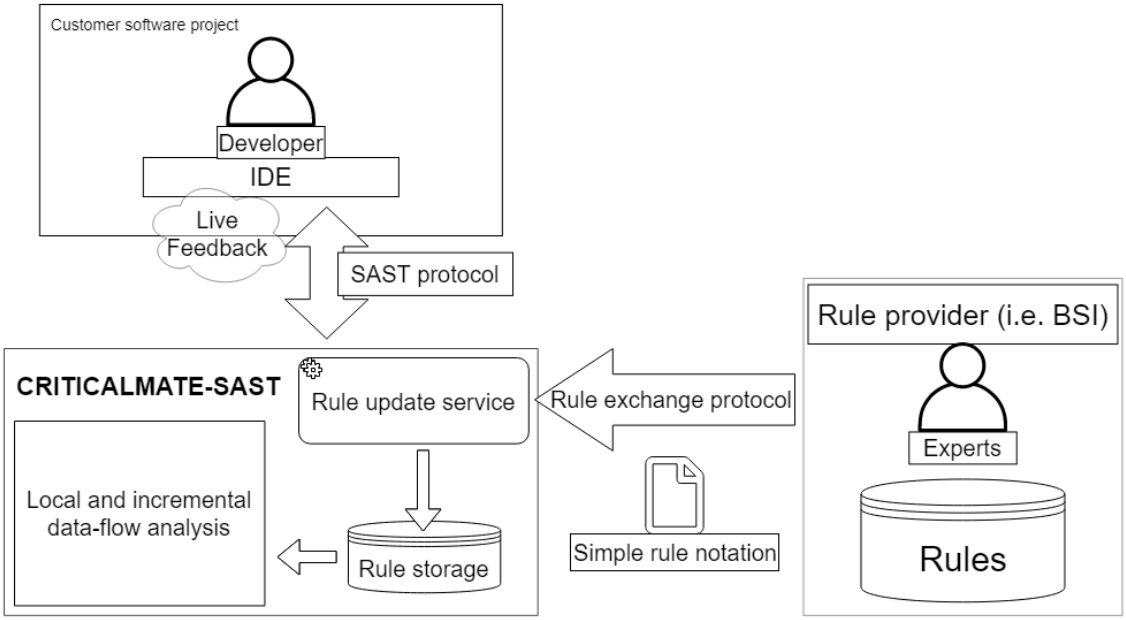

Figure 2.1: Components of criticalmate SAST [RWWM19]

3 Basics

3.1 Standards

In the following sections, the relevant standards for data security and cryptographic procedures are presented. For the cryptographic procedures, mainly the recommendations of the BSI are considered, but also international standards from the National Institute of Standards and Technology (NIST) are taken into account.

BSI TR-02102-1 The technical guideline TR-02102-1 from the BSI provides guidance for the use of cryptographic mechanisms. It contains recommendations for the algorithms and protocols to use in cryptographic procedures and the associated secure key lengths [Inf20a].

BSI TR-02102-2 In the technical guideline TR-02102-2, the BSI provides information and recommendations for using the Transport Layer Security (TLS) protocol. It contains guidance for the selection of the protocol's version, the securest algorithms, the cipher suites as well as the appropriate key lengths [Inf20b].

NIST SP 800-52 The NIST Special Publication 800-52 contains guidelines for the use of the TLS protocol. In detail, it provides guidance for the selection of version and configuration of the TLS protocol for both client and server. Therefor, only NIST approved cryptographic procedures and algorithms are acceptable [MC19].

3.2 Definition of "data security"

As the focus of this work is on building a secure rule exchange infrastructure, the security of exchanged data is of utmost importance. Data security basically consists of

- authenticity,
- confidentiality and
- integrity.

In this context, authenticity means that a communication actually takes place with the desired communication partner and not someone else who impersonates them. Data confidentiality is used to make sure nobody except the sender and recipient is able to read the actual content of exchanged data. As the third and last part, data integrity is about the trustworthiness of exchanged data. Data can be considered trustworthy, if there is a way to ensure the data has not been changed on the way to the recipient.

3.3 Interface design rules

In his book "Designing the User Interface: Strategies for Effective Human-Computer Interaction" from 1998 [Shn98], the computer scientist and professor Ben Shneiderman reveals 8 golden rules of interface design, which are still valid and important today. Products of very successful companies like Google, Amazon, Microsoft and Apple reflect these highly important rules. As they are indispensable when attempting to build good interfaces, they will be presented and explained in the following.

- **Strive for consistency** by using the same color scheme, designs and positions all over the interface for buttons, input forms, dialogs, menus and other elements. Also the same terminology in dialogs, feedbacks and especially in important notifications (success or error) should be used. When doing this the correct way, users will be able to perform new actions faster and quickly become familiar with the interface.
- **Enable frequent users to use shortcuts**. When providing shortcuts, experienced users can perform actions without always switching between keyboard and mouse and thereby speed up the completion of their tasks.
- **Offer informative feedback**. The user should always be informed about the state of the interface. For every action, the user should get an informative and human-readable feedback.
- **Design dialog to yield closure**. After completing an action, the user should get an immediate feedback informing him about the effect of this action.
- **Offer simple error handling**. Interfaces and systems should be designed in a way that it is hard to make mistakes and errors are unlikely. But if an error occurs, the user should get a comprehensible feedback about what exactly happened and how it can be resolved.
- **Permit easy reversal of actions**. The interface should provide easy ways for users to undo their actions.
- **Support internal locus of control**. Users should always be the initiators of actions and not just the responders. This gives users the feeling that they are in charge of the system and have the full control.
- **Reduce short-term memory load**. Attention can be seen as a limited resource and humans are only able to keep about five items at a time in the short-term memory. So, users should not have to recall large amounts of information while performing their tasks.

4 Requirements engineering

Before starting with the actual requirements engineering, the rule exchange infrastructure could be divided into the following three components:

- The rule exchange server component, which runs with each rule provider
- The rule exchange client component, which is included in a SAST tool
- The administration component, which provides a GUI that can be used by rule providers to manage their rules

For the requirements engineering, first of all the requirements for the rule exchange server and client were identified, as these two components form the actual rule exchange part in the infrastructure. Some existing data exchange infrastructures and protocols were analyzed so that requirements could be extracted. Also, some could be identified with the help of the project description of the criticalmate project ([RWWM19]).

Then, based on the requirement that rule providers should have a GUI which enables them to perform CRUD operations on rules, the requirements for the administration component could be identified. As the last step, requirements from the task described in section 1.3 were included and a brainstorming was carried out to complete the requirements for each component.

To ensure the resulting infrastructure complies with the latest security recommendations, the guidelines described in section 3.1 were taken into account while defining the security requirements for each component.

During this process a total of 53 functional and non-functional requirements could be identified and defined. These resulting requirements will be presented in the following. The word “user” will be used as synonym to “member of a rule provider organization” and an “administrator” is a special kind of user owning more privileges.

4.1 Functional requirements

The following functional requirements specify the features and functions each component has to provide.

4.1.1 Rule exchange server component

[R01] Rule creation
The server must offer an interface to allow external systems to create rules.

[R02] Rule retrieval
The server must offer an interface to allow external systems to retrieve all rules.

[R03] Rule editing
The server must offer an interface to allow external systems to edit the content of rules.

[R04] Rule deletion
The server must offer an interface to allow external systems to delete rules.

[R05] Authentication
The server must offer an interface to allow external systems to authenticate.

[R06] Authorized requests
The server must ensure that requests from external systems, which require manipulation of one or more rule(s) in the data storage ([R01], [R03] and [R04]), are authorized.

[R07] Authorization
The server must provide a way to check whether an external system is authorized or not.

[R08] Response data format
The server must offer its information specified in [R02] to external systems in the JavaScript Object Notation (JSON) and Extensible Markup Language (XML) format.

[R09] Successful requests
The server must inform an external system if its request could be processed successfully.

[R10] Unsuccessful requests
The server must inform an external system if its request could not be processed successfully.

[R11] Error response
The response from the server to the event specified in [R10] must provide some information to the external system about the reason why its request could not be processed properly.

[R12] User creation
The server must offer an interface to allow external systems to create user accounts.

[R13] User retrieval
The server must offer an interface to allow external systems to retrieve all existing user accounts.

[R14] User editing
The server must offer an interface to allow external systems to edit user accounts.

[R15] User deletion
The server must offer an interface to allow external systems to delete user accounts.

[R16] Account manipulation
The server must ensure that an external system, which requests the manipulation or retrieval of one or more user account(s) ([R12] - [R15]), is authorized to do so. In more detail, this also means that there must be a way to distinct users of external systems, which have the permission to perform CRUD operations on user accounts (“administrators”) from users, which only have permission to perform these operations on rules (“default users”).

[R17] Own account manipulation
The server must offer an interface to allow authenticated users of external systems to edit their own account. This means that the server has to allow authenticated users to edit at least their own name and log in credentials.

[R18] Lost update
The server must provide a functionality to detect whether an update in data storage would overwrite a newer update (lost update problem).

4.1.2 Rule exchange client component

[R19] Rule download
The client must offer an interface to allow SAST tools to download rules from a specific rule provider.

[R20] Rule storage
The client must provide a way to store downloaded rules in a persistent local data storage.

[R21] Retrieve rules
The client must offer an interface to allow SAST tools to retrieve the downloaded and locally stored rules in an appropriate format.

4.1.3 Administration component

[R22] Log in
The administration component must offer a functionality for users with an user account to log in and thereby authenticate to the administration component as well as the rule exchange server.

[R23] Unauthenticated access
The administration component must prohibit the access to any functionality for users, who are not logged in and thereby are not authenticated.

[R24] Existing rules
The administration component must offer an overview of all existing rules for authenticated users.

[R25] Rule creation
The administration component must offer a functionality for users to create and then add a new rule to the rule storage.

[R26] Rule editing
The administration component must offer a functionality for users to edit an existing rule.

[R27] Rule deletion
The administration component must offer a functionality for users to delete an existing rule from rule storage.

[R28] Search rules
The administration component must offer a functionality for users to search for specific rules.

[R29] Sort rules
The administration component must offer a functionality for users to sort rules according to different categories.

[R30] Existing users
The administration component must offer an overview of all existing users for administrators.

[R31] User creation
The administration component must offer a functionality for administrators to create and then add a new user account to the data storage.

[R32] User editing
The administration component must offer a functionality for administrators to edit an existing user account.

[R33] User deletion
The administration component must offer a functionality for administrators to delete an existing rule from rule storage.

[R34] Search users
The administration component must offer a functionality for administrators to search for specific users.

[R35] Sort users
The administration component must offer a functionality for administrators to sort users according to different categories.

[R36] Own account manipulation
The administration component must offer a functionality for users to edit their own account information.

[R37] Lost update
The administration component must not allow a user to update an entity if he thereby would overwrite a newer update (lost update problem).

4.2 Non-functional requirements

The following non-functional requirements specify the qualities each component has to provide.

4.2.1 Rule exchange server component

[R38] Data storage
The server must store the rules and user accounts in a persistent database.

[R39] Flexibility and portability
The server component has to run with different rule providers on different infrastructures and platforms. Therefor, the server must not be bound to a specific web server from a specific manufacturer, it must be able to run with different web servers. Also, for flexibility, rule providers should be able to configure the server to use an existing database. Therefore, the server does not have to have its own database, which would be superfluous.

[R40] Security
The server must ensure that data exchange with external systems only happens in a secure way. In more detail, this means that the aspects of data security described in section 3.2 must be adhered to.

[R41] Expandability
The server must be expandable with reasonable effort. If there are new requirements the server has to meet, it should be possible to implement them fast and without great difficulties.

[R42] Maintainability
The server must achieve a high level of maintainability. This does especially mean that the correction of defects or their cause or the change of the environment should not be difficult tasks. In the end, this should lead to a high level of reliability. Amongst other things, it can be achieved by commenting the code well and extensively.

[R43] Programming language
The server component must be implemented in the programming language Java.

4.2.2 Rule exchange client component

[R44] Portability
The client component has to be able to run on recommended versions of all common operating systems. The systems which have to be supported necessarily are Microsoft Windows, Linux and macOS.

[R45] Security
The client must ensure that the security goals specified in section 3.2 are adhered to. In more detail, this means that the client must only accept a set of rules that he can be sure of it originates from the rule provider he wants to download it from (authenticity) and it was not changed by someone else (integrity).

[R46] Expandability
The client must be expandable with reasonable effort. If there are new requirements the client has to meet or if the server offers new functionalities the client has to use, it should be possible to implement them fast and without great difficulties.

[R47] Maintainability
The client must achieve a high level of maintainability. This does especially mean that the correction of defects or their cause or the change of the environment should not be difficult tasks. In the end, this should lead to a high level of reliability. Amongst other things, it can be achieved by commenting the code well and extensively.

[R48] Programming language
The client must be implemented in the programming language Java.

4.2.3 Administration component

[R49] Portability
The administration component will be run by many different users on different platforms. To achieve platform independence, a high level of flexibility and because no client installation is needed, this component will have to be realized as a web application.

[R50] Web browser
The web application must be accessible on all modern web browser, especially on Google Chrome, Opera, Safari, Mozilla Firefox and Microsoft Edge.

[R51] Usability
The administration component must provide a high level of usability. Therefor, especially the 8 golden rules for user interface design described in section 3.3 must be followed. Also, to make it understandable for everyone, the language of the GUI must be english.

[R52] Expandability
The administration component must be expandable with reasonable effort. If there are new requirements it has to meet or if the server offers new functionalities the client has to use, it should be possible to implement them fast and without great difficulties.

[R53] Maintainability
The administration component must achieve a high level of maintainability. This does especially mean that the correction of defects should not be difficult. In the end, this should lead to a high level of reliability. Amongst other things, it can be achieved by commenting the code well and extensively.

5 Specification

In this chapter, the specification of the rule exchange infrastructure is presented. The concepts, techniques and design decisions used in the three components are explained and justified. All of this follows the requirements specified in chapter 4.

5.1 Server component

When reviewing the requirements for the rule exchange server, it leads to the result that the main task for this component is to perform the CRUD operations for persistent storage. Because of this, the right choice of data exchange pattern as well as persistent storage is crucial. A data exchange pattern consists of the following three components [CR20]:

- an architectural pattern,
- a data format and
- a communication protocol

Years ago, mainly because of the lack of reliability and capacity, applications exchanged their data using files. Today, the trend is towards web services using synchronous, request-response and message-based communication [CR20].

To ensure every SAST tool supporting the rule exchange infrastructure is able to access the rules from any rule provider over the internet, the server component will have to be a web service. This service will offer an Application Programming Interface (API) to enable the client as well as the administration component to perform the CRUD operations for persistent storage. In particular, building the server according to an API design pattern drastically increases flexibility, expandability and maintainability. This is mainly because interfaces can be added, changed or deleted with comparatively little effort and changes can be made to the backend application without the need to also make changes to interfaces.

5.1.1 Architecture

Currently, there are three common and popular types of web services for the API pattern [CR20]:

- GraphQL
- SOAP
- Representational State Transfer (REST)

Starting with GraphQL, this technology basically is a data query and manipulation language, but also a runtime to provide interfaces external applications can use. GraphQL offers added value compared to other systems when dealing with many complex entities that reference each other. GraphQL is able to take complex queries, collect all data needed from storage to answer the query and return it. This makes data aggregation from multiple sources very easy for an external system, because it only has to use one "smart" endpoint instead of many [Foua]. But looking at the requirements, the infrastructure does not have these complex data structures needed to justify the additional effort of implementing GraphQL. Here, the use of GraphQL only would make the queries more complex without providing (notable) advantages over the remaining two other technologies. These arguments speak against the use of GraphQL, at least for this particular case.

On the one hand, the network protocol SOAP can be used to exchange data between web systems and on the other hand to perform a Remote Procedure Call (RPC). As data format, SOAP primarily uses XML and it can be used with several communication protocols including Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP) and File Transfer Protocol (FTP).

REST, however, is more of an architectural style than just a simple protocol. It defines many guidelines for a flexible implementation. For example, as the name suggests, statelessness and the use of a uniform application interface. REST can be used in combination with the communication protocol HTTP and the data format JSON, but it is by no means restricted to these techniques.

It must now be decided whether to use a REST architecture as the web service or implement the SOAP protocol. It is problematic if not impossible to directly compare a protocol and an architecture. However, it is possible to weigh the general advantages and disadvantages of a REST architecture and the ones of the SOAP protocol for the rule exchange infrastructure against each other and make a decision based on that.

One of the main differences between REST and SOAP is the degree of coupling between client and server. A SOAP client is tightly coupled to the server and there's a strict contract between them [ML07]. So, the client needs prior knowledge of the way the server works. A REST client on the other hand is rather loosely coupled and, except for the entry Uniform Ressource Identifier (URI), there's no need to have prior knowledge, because the resources return links the client can follow. This basic principle of REST is known as Hypermedia as the Engine of Application State (HATEOAS). It was already described by Roy Thomas Fielding in his dissertation from 2000, in which he proposed, among other things, this (for this time) new architectural style [Fie00].

> "A REST API should be entered with no prior knowledge beyond the initial URI . . . From that point on, all application state transitions must be driven by the client selection of server-provided choices . . . " - Roy Thomas Fielding, computer scientist [Fie08]

When taking the non-functional requirements flexibility, expandability and maintainability into account, this speaks in favor of using REST for the rule exchange server.

Next, the XML data format used with SOAP is much more complex than the JSON format used with REST. This is perfect when performing complex queries with complex data structures, but for simple queries the transmission volume overhead is too big. Furthermore, an XML document has to be built and validated on the client side and then validated and parsed on the server side, which is why the computational effort is much greater here. In addition to that, the creation of SOAP services from scratch is much more complex than the creation of REST services. In general, REST APIs are lean and as a result of that, they perfectly fit in today's modern Internet of Things. REST

provides a higher level of reliability as well as fault tolerance, what manly relies on the statelessness. Because state does not matter, services can be restarted quickly and the communication between them is very low, which is why it can be scaled horizontally very well.

When weighing the security of REST against SOAP, REST provides great advantages for the rule exchange infrastructure. REST APIs consist of multiple endpoints and each of them is addressable via a Unique Ressource Locator (URL). As a result of that, administrators can block access to specific URLs from outside the company network using the firewall. This is important in this case, as when looking at the requirements it is noticeable that the server and the administration component both run with the rule provider. This could now block access from outside its network to the URLs only used by the administration component (which are, at least by default, all except the one to retrieve all rules), which results in a drastic increase of security.

Ultimately, all of the above reasons speak in favor of using a REST architecture and against using the SOAP protocol. JSON is very compact, fast, readable and therefore very intuitive, and so it will be used as data format for communication. Fielding already said that "REST does not restrict communication to a particular protocol ..." [Fie00]. Usually the HTTP protocol is used and as it opens the possibility to use self-describing HTTP methods (like for example the GET-method) and answers containing HTTP status codes, it will be used as communication protocol for the rule exchange infrastructure.

5.1.2 Interfaces

The endpoints of the REST API are the interfaces external systems can use. All interfaces needed to provide the required functionality can be extracted from the functional requirements described in section 4.1. Each consists of an HTTP method that must be used and a URL. In a route, the *{id}* parameter must be replaced with the Universally Unique Identifier (UUID) of the resource to be accessed. In the following, the endpoints, the expected input parameters and the responses are specified.

To create a rule, a PUT request has to be sent to the endpoint "*/rules/create*". In the HTTP body of the request, there must be a valid JSON string containing two fields: a *name* field containing the name and a *content* field containing the content of the rule to create. The server will answer with the status code "*200 (OK)*" if the rule was successfully created. Otherwise, the status code will be "*400 (Bad Request)*" to state that the rule could not be created due to a missing field.

To retrieve all rules, a GET request has to be sent to the endpoint "*/rules*". As this is the main endpoint used by SAST clients, the HATEOAS principle is supported to keep the coupling loose and thus the server will answer with an HTTP response object containing a string in JSON Hypertext Application Language (HAL) format. Whenever a rule provider changes the route of this endpoint, the response of the request made to "*/rules*" contains the URL from which the rules can be fetched as the *all* property. It should be noted that there is no need for the other endpoints to support the HATEOAS principle, as they are only used by the administration component which runs with the rule provider. Therefore, it is not a disadvantage that the coupling is a little tighter. Usually, however, the response contains a list of all rules as embedded resource like shown in listing 5.1. With the link specified for each rule in the list and by using a GET request, a single rule can be fetched from the server.

Listing 5.1 JSON HAL response

```
{
  "_embedded": {
    "ruleList": [{
      "id":"68d70ce0-5ab4-4e5d-9bfa-33473930ffa9",
      "name":"Rule 1",
      "content":"Content of Rule 1...",
      "version":12,
      "creationDate":"18.02.2021",
      "_links": {
        "self": {
          "href":"https://localhost:8443/rules/68d70ce0-5ab4-4e5d-9bfa-33473930ffa9"
        }
      }
    }]
  },
  "_links": {
    "self": {
      "href":"https://localhost:8443/rules"
    },
    "all": {
      "href":"https://localhost:8443/rules"
    }
  }
}
```

To edit a rule, a POST request has to be sent to the endpoint "*/rules/update*". In the HTTP body of the request, there must be a valid JSON string containing three fields: an *id* field containing the UUID of the rule to update, a *name* field containing the new name and a *content* field containing the new content of the rule. The server will answer with the status code "*200 (OK)*" if the rule was successfully updated. Otherwise, the status code and the error message of the response provide information about what happened and how to solve the problem.

To delete a rule, a DELETE request has to be sent to the endpoint "*/rules/delete/{id}*". The server will answer with the status code "*200 (OK)*" if the rule with the UUID provided in the URL was successfully deleted. Otherwise, the status code will be "*404 (Not Found)*" stating that a rule with the given UUID could not be found in the database.

To retrieve the account information for the own account (for which the given JSON Web Token (JWT) was issued), a GET request has to be sent to the endpoint "*/account/information/get*". The server will answer with a user object in JSON format which looks like the example provided in listing 5.2. The fields *id*, *firstName*, *lastName* and *email* are self-explanatory. The field *role* contains a string specifying the role of the user, which can either be "default" or "administrator". The field *locked* contains either *0* if the account isn't locked or the time until the account is locked as milliseconds since midnight, January 1, 1970 Universal Time Coordinated (UTC), as this is also used by Unix epoch and generally easy to convert. The *loginAttempts* field states how many failed

Listing 5.2 User object in JSON format

```
{
  "id":"6a9d47d0-6161-44ca-bdf0-61fadd895695",
  "firstName":"Test",
  "lastName":"User",
  "role":"default",
  "email":"testuser@email.com",
  "locked":"0",
  "loginAttempts":"0",
  "version":"2"
}
```

login attempts were made to this user account and the meaning of the *version* field is explained in section 5.1.3. If something goes wrong, the status code and the error message of the response provide information about what happened and how to solve the problem.

To update the own account information, a POST request has to be sent to the endpoint "*/account/information/update*". The request body has to contain four fields formatted as JSON: a *firstName* field containing the new first name, a *lastName* field containing the new surname, an *email* field containing the new email address for the account and a *version* field containing the row version like explained in section 5.1.3. The server responds with the status code *200 (OK)* if the account information were successfully update or with an error status code and message providing information about what exactly went wrong.

To update the own account password, a POST request has to be sent to the endpoint "*/account/password/update*". The request body has to contain two fields formatted as JSON: an *oldPassword* field containing the old password from the account and a *newPassword* field containing the new password. The server responds with the status code *200 (OK)* if the account password was successfully update or with an error status code and message providing information about what exactly went wrong.

To create a user account, a PUT request has to be sent to the endpoint "*/admin/users/create*". In the HTTP body of the request, there must be a valid user object in JSON format containing the fields *firstName*, *lastName*, *role*, *email* and *password*. The server will answer with the status code *200 (OK)* if the account was successfully created. Otherwise, the status code will be *400 (Bad Request)* to state that the user could not be created due to a missing field or the fact that a user account with the given email address already exists.

To retrieve all users, a GET request has to be sent to the endpoint "*/admin/users*". The response contains a list of all user accounts (see listing 5.2) as JSON format. To retrieve a single user account by its UUID, the endpoint "*/admin/users/{id}*" can be used by sending a GET request.

To edit a user account, a POST request has to be sent to the endpoint "*/admin/users/update*". In the HTTP body of the request, there must be a JSON user object looking like the example shown in listing 5.2, but with an additional *password* field containing the new password for this account. The server will answer with the status code *200 (OK)* if the account was successfully updated. Otherwise, the status code and the error message of the response provide information about what happened and how to solve the problem.

To delete a user account by its UUID, a DELETE request has to be sent to the endpoint "*/admin/users/delete/{id}*". The server will answer with the status code *200 (OK)* if the account with the UUID provided in the URL was successfully deleted. Otherwise, the status code will be *404 (Not Found)* stating that a rule with the given UUID could not be found in the database.

Last but not least, to authenticate to the server a POST request can be sent to the endpoint "*/login*". The request body has to contain two fields: an *email* and a *password* field. The login procedure is explained in detail in section 5.5.2.

All these endpoints are to be used by the administration component and therefore require authorization like explained in section 5.5.3. Exceptions to this are the endpoints to retrieve rules, which should be used by SAST tools to download rules from a specific rule provider, and the login route.

5.1.3 Lost update prevention

Like shown by the sequence diagram in figure 5.1, it can happen that several employees of a rule provider edit the same entity (rule or user account) at the same time in the administration component. This could work in a similar way as described below. Firstly, Client 1 downloads the newest version of the entity which should be edited from the server (step 1 and 2). While the user edits this entity, Client 2 downloads the newest version of the same entity from the server (step 3 and 4) and the user starts to edit it. After the user of Client 1 is done editing the entity, the updated entity is sent back to the server (step 5) which stores the update in the database. After that happened, also the user of Client 2 is done editing the entity and also sends it back to the server (step 7). The server stores the update in the database and thus overwrites the previous update made by Client 1. In the end, the update made by Client 1 is completely lost and only the update made by Client 2 is stored in the database. This problem is known as the lost update problem.

To prevent this, a mechanism known as Optimistic Locking is used, as it is very hard if not impossible to implement a reliable Pessimistic Lock in a stateless client/server application. When using Pessimistic Locking, a client needs to acquire a lock before the user can start editing an entity. After the editing has completed, the client returns the lock to the server which can then pass it to someone else. This ensures that only one client processes an entity at a time, but whenever a user closes the web application while editing an entity, the lock can not be returned. This problem could be solved by using a timeout for locks, but it would have to be large enough to give users much time to edit an entity. In the end, it still would be possible that resources are locked without anyone editing them. When using the Optimistic Locking approach on the other hand, each entity gets a version attribute which is a simple numeric value. If the entity is updated in the database, the version is incremented. That solves the lost update problem, because if two clients successively update an entity, the version is incremented after the first update. If the second client tries to update the entity, the server detects that the version of the entity that was received is smaller than the version of the entity from the database. So, the second update is rejected and the client gets informed about that. To perform the update, the first thing to do is to download the latest version of the entity. After that, changes can be applied and it can be sent back to the server which updates the database.

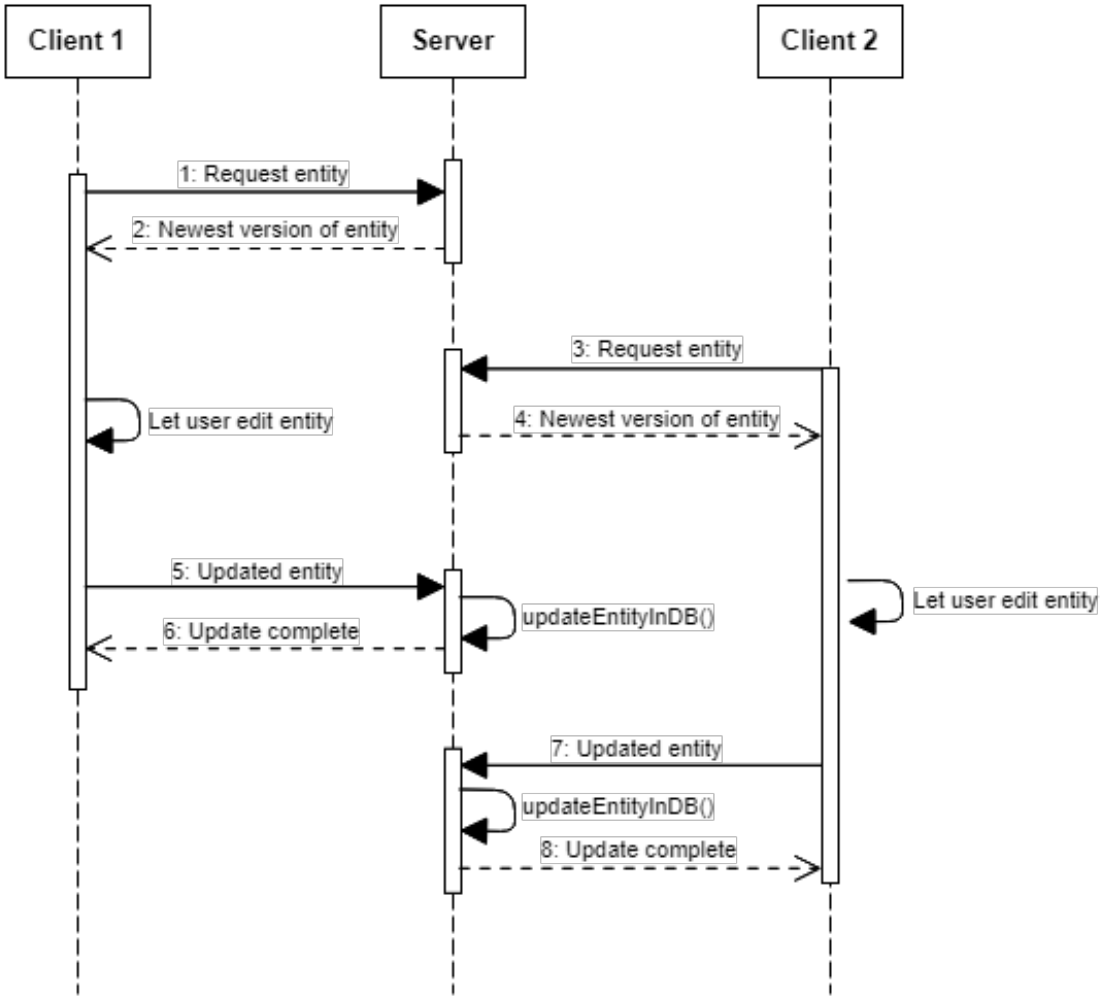

Figure 5.1: Example for a lost update when editing entities

5.1.4 Data storage

As persistent data storage, a relational database should be used. Because the Relational Database Management System (RDBMS) is not directly part of the server application itself, there are many possibilities when it comes to choosing a database. PostgreSQL and MySQL are among the best and most popular RDBMS supporting the Structured Query Language (SQL). Since PostgreSQL is even an Object-Relational Database Management System (ORDBMS), it perfectly harmonizes with the object-oriented programming language Java and should be used with preference. Nevertheless, each rule provider company can decide for itself which RDBMS it wants to use.

5.2 Client component

Taking into account the requirements for the client component, the main tasks are to download the rules from rule providers, store them in a local persistent storage and make them accessible for the SAST tool. To achieve a high level of portability, the REST client will be implemented using the programming language Java, as it is platform-independent. Since expandability and maintainability are also important requirements for the client, it will be implemented as a library providing interfaces for the SAST tool to download, store and load rules. Therefore, the coupling between client and SAST tool is not too tight and it is, for example, possible to change the client's underlying implementation without touching the interface and thus without having to adjust code of the SAST tool.

5.2.1 Rule storage

The client has to be able to store the rules of each rule providers in a persistent way. As with the server, a RDBMS should be used here. However, an external RDBMS would be superfluous and because of that, a lightweight embedded in-memory database should be used. One of the best and fastest databases that meets all requirements is the H2 RDBMS. It supports queries using SQL and when used as in-memory database, only one database file is needed to store the data. It is embedded in and completely managed by the client application and as a result of that, no additional application or setup is needed to use the rule exchange client.

5.2.2 Interfaces

The client library provides several interfaces, which can be divided into the following two categories:

- Methods for downloading rules from a rule provider specified using an Internet Protocol (IP) or Domain Name System (DNS) address. These methods use the interfaces to retrieve rules from the server and return the data that were received as a list of rules.
- Methods to load and store rules in a (specific) table in the local in-memory database.

5.3 Administration component

When considering the requirements, the administration component is responsible to provide a GUI for rule providers, which enables them to perform the CRUD operations for rules. Furthermore, there should be a distinction between default users and administrators, whereby administrators should have the possibility to also perform CRUD operations on user accounts. The requirements also want this component to be a web application, which has many advantages including that no client installation is needed and that the portability is very high.

The web application provides a login functionality with which employees from a rule provider, for whom a user account exists in the backend database, can authenticate to the server. A user enters its email address as well as a password and, by using the login interface from the server, the

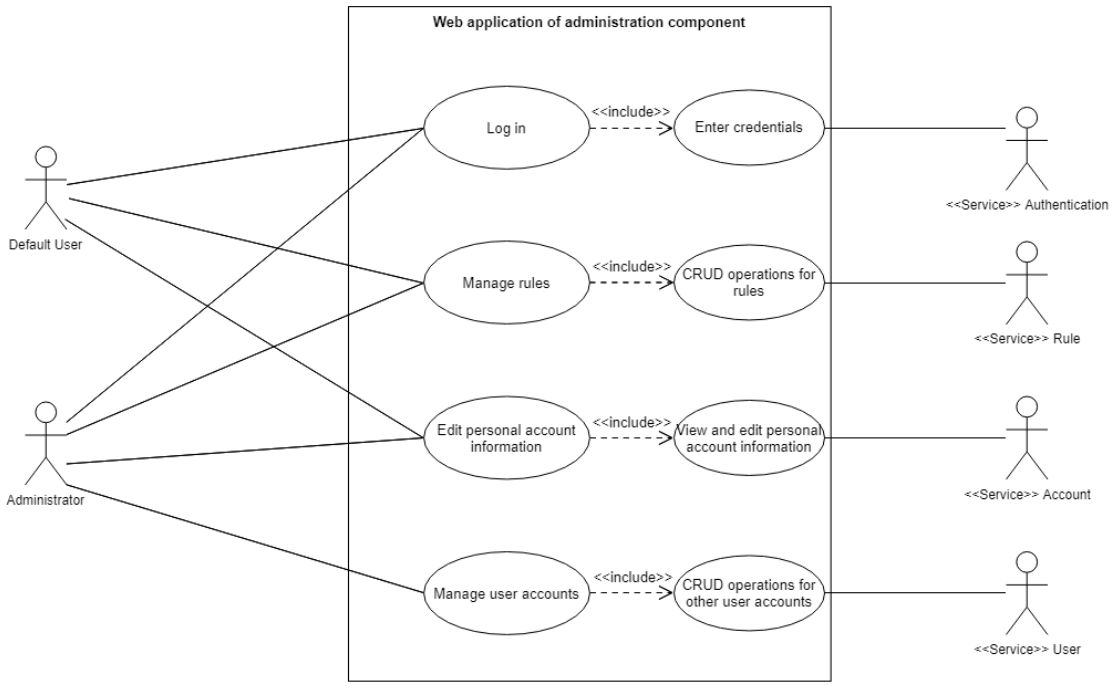

Figure 5.2: Use case diagram of administration component

email/password combination is validated and the user is possibly granted access. For authenticated users, the REST endpoints described in section 5.1.2 are used to perform CRUD operations for rules and, if the user is an administrator, for user accounts.

5.3.1 User interface

As a first draft, the rough layout of the user interface for the administration component was drawn as a paper prototype. Therefore, the GUI design rules from Shneiderman described in section 3.3 were followed. In addition to that, guidelines and components of Material Design developed by Google LLC were used. Using the Material Design ecosystem for the prototype and also for the later implementation offers many advantages. It provides a high level of flexibility, as often specific decisions in the way to implement the design are left up to the designer. The main components of the design have a card-like appearance and due to the physical depth effect mostly caused by shadows, users immediately recognize important areas and things they can interact with. All in all, Material Design is very user-friendly, intuitive and characterized by elegant minimalism.

While designing the GUI, it could be divided into the following 5 main components:

- The login page is used to authenticate a user to the server. It consists of a material card containing input fields for an email address and a passwords as well as a button to submit the login information.

- The account settings page enables users to edit their account information. It consists of a material card containing input fields with which a user can edit his first name, surname and email address. For security reasons, to change the password the new password has to be entered twice and the old password also has to be provided to confirm the identity of the user. To keep the user interface clear, the account settings page contains a button which opens a dialog to handle the process of changing the password.

- The welcome page acts as landing page for the web application after login.

- The rule page basically consists of a table containing all the rules created by this rule provider. Every row in this table contains the name and creation date of a rule as well as buttons to edit or delete this rule. At the bottom of the page, there is a Floating Action Button which, according to Material Design, opens a dialog to enable the user to create a new rule.

- The user account page is only accessible for administrators. Similar to the rule page, it consists of a table containing all the user accounts registered for this rule provider. Every row in this table contains the first name, surname and role of a user as well as buttons to edit or delete the account. At the bottom of the page, there is a Floating Action Button which, according to Material Design, opens a dialog to enable the administrator to create a new user account.

In addition to the elements just described, there are two more elements on each page except the login page: A Toolbar and a Sidenav. The Toolbar is located at the top of the screen and extends across its full width. It contains the name of the application and a button group to access the account settings and to log out. The Sidenav, on the other hand, is located at the left edge of the screen and extends across its full height. It contains buttons which enable a user to navigate between the welcome, rule and user account page. Both of these elements are designed according to Material Design guidelines. To provide a brief overview, the use case diagram 5.2 shows the basic functionalities of the GUI as well as the allocation of frontend services.

5.4 Architecture

Now that all of the three main components of the rule exchange infrastructure have been described, figure 5.3 gives a brief overview on how they interact with each other. On the infrastructure of the rule provider, on the one hand there is the server component including the REST API, the application logic and the database. On the other hand, there is the web application forming the administration component. Outside the network of the rule provider company, there is the SAST tool containing and using the rule exchange client as library. The client uses the REST endpoints offered by the server to download rules required for the SAST tool to perform the analyzes.

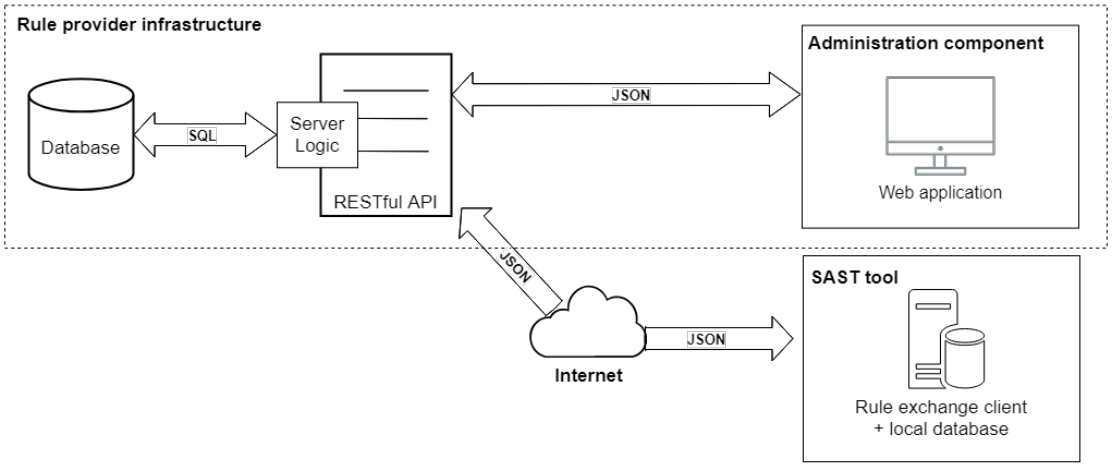

Figure 5.3: Components and interactions of the rule exchange infrastructure

5.5 Security concept

As already mentioned, security is of utmost importance for the whole infrastructure. It is therefore essential to have a good security concept and to use a strong encryption protocol for message exchange. The security concept and especially the encryption protocol used by the rule exchange infrastructure is presented and explained in the following sections.

It should be noted here that the terms *authentication* and *authorization* are not synonyms, as is often wrongly assumed. In the world of identity and access management they have different meanings. In any security process *authentication* is the first step and describes the process of ascertaining that a user is who they pretend to be. In the context of the rule exchange infrastructure, a user authenticates to the administration component and server by entering a valid email address/password combination. *Authorization*, however, takes place after the successful authentication and describes the process of granting a user permission to one or more resources. In the context of the rule exchange infrastructure, a user is authorized to perform CRUD operations on rules after a successful login. Administrators also are authorized to manage user accounts.

5.5.1 Encryption

The TLS protocol can be used to securely exchange data in insecure networks, where especially the security objectives described in section 3.2 must be adhered to [Inf20b]. For the data exchange between server and administration component as well as between server and rule exchange client (SAST tool), the latest and most secure version 1.3 of the TLS protocol like specified by the Internet Engineering Task Force (IETF) is used [Res18]. Therefore, the server ensures that only Hypertext Transfer Protocol Secure (HTTPS) connections that use TLS 1.3 are allowed and that HTTP connections are rejected.

In the TLS protocol, a so-called *cipher suite* defines the specific algorithms used to secure a connection. In the case of TLS 1.3, a cipher suite consists of two components: An authenticated encryption algorithm to ensure Authenticated Encryption with Associated Data (AEAD) and a hash

Listing 5.3 Example for body of the login request

```
{
  "email":"users-email@address.com",
  "password":"user_password"
}
```

algorithm for key derivation. The cipher suite *TLS_AES_256_GCM_SHA384* is one of the three cipher suites recommended for TLS 1.3 by the BSI and will be used here, as it includes the longest key lengths [Inf20b]. This cipher suite uses the symmetric Advanced Encryption Standard (AES) encryption algorithm with a key of the length 256 bit to encrypt and decrypt exchanged data. To provide AEAD, the Galois/Counter Mode (GCM) is used as the mode of operation and to derive keys Secure Hash Algorithm 384 (SHA-384) is used.

Basically, the communication between either rule exchange client and server or administration component and server can be divided into two phases: the TLS handshake and the TLS record. The main challenge during the TLS handshake is to agree on a shared symmetric AES key and to make sure nobody but the client and the server knows this key. Therefore, the Diffie-Hellman key exchange (DHE) is used, with which the two communication parties are able to securely choose and exchange a secret session key [RDM08]. After client and server agreed on a session key, the server sends his encrypted X.509 certificate to the client to confirm its identity and especially to rule out a man-in-the-middle attack. After the client verified the certificate and thus the identity of the server, the establishment of a secure HTTP connection is complete [Res18].

TLS record then uses the during handshake negotiated session key to encrypt the HTTP connection using AES-256 in Galois/Counter mode to ensure confidentiality, integrity and message authentication like recommended by official bodies including BSI ([Inf20b]) and NIST ([MC19]). In the end, an attacker has no chance of knowing the secret session key and therefore cannot read or tamper the exchanged information.

5.5.2 Authentication

To authenticate to the server, an external system (like the administration component's web application) sends an HTTP POST request to the */login* endpoint of the server. The HTTP body of this request has to contain an *email* and a *password* field in JSON format to provide the email address and password of the user like in the example provided by listing 5.3.

After receiving the request, the server checks whether the email address belongs to an existing user account and whether the given password is correct. If one of these two actions fails, the server responds with the HTTP status code "*401 Unauthorized*" and a message stating that the email/password combination is invalid. Due to security reasons, the response does never contain information about the existence of the user account with given email address. Another security measure is that as soon as an incorrect password is entered for an account five times in a row, the accounts gets automatically locked for an hour.

Listing 5.4 Example for the body received after successful authentication

```
{
  "body":
  {
    "id":"78cfdcb8-819f-11eb-8dcd-0242ac130003",
    "firstName":"Test",
    "lastName":"User",
    "email":"test@user.de",
    "role":"DEFAULT",
    "locked":0,
    "jwt":"..."
  }
}
```

However, if the combination of email and password is correct, the server responds with the status code “200 OK”. The JSON formatted body of the response contains information about the user and a newly generated JWT needed for authorization. Listing 5.4 provides and example for the layout of this body.

These information, especially the JWT, should be stored in a secure local storage, like the local storage of a browser in the case of a web application.

5.5.3 Authorization

After the authentication process described in section 5.5.2, the external system is authenticated and in possession of a JWT. Basically, JSON Web tokens consist of the following three parts separated by dots:

- Header
- Payload
- Signature

The header is a Base64Url encoded string and contains the type of the token, in this case JWT, and the signing algorithm that was used for the signature part, in this case Keyed-Hash Message Authentication Code (HMAC) with a length of 512 bits. The payload is also an Base64Url encoded string containing some additional data, called claims. The most important claims are:

- The subject, which contains the email address of the user for whom the JWT was issued.
- The expiration time, which contains the time stamp when the JWT expires and thus becomes invalid. In the case of the rule exchange infrastructure, the default lifetime of a JWT is set to 15 minutes.
- The role, which contains the name of the role the user for whom the JWT was issued has (“admin” or “default”).

Up to this point the JWT would be useless, as every attacker is able to issue a valid JWT and request authorization from server. Therefore, the JWT contains the third and last part called signature. To create the signature, the Base64Url strings of the header and payload are concatenated. A signature is then calculated from this using a secret key that only the server knows and the signing algorithm specified in the header (HMAC512 in this case). Thereby, only the server is able to issue valid JWTs [JBS15].

If an external system now wants to access a protected resource, it has to prove to the server that it is authorized to do so. This is done using the JWT. For each request that requires authorization (which are all except the ones to retrieve rules), the requesting system has to provide its JWT in the *Authorization* field of the HTTP request header according to the Bearer schema. The *Authorization* header of the request has to have the following layout, with "<token>" replaced with the JWT:

```
Authorization: Bearer <token>
```

Whenever the server receives a request that requires authorization, the JWT is taken from the header and an attempt is made to validate it. If the token is invalid (for example because of an invalid signature or due to expiration), the server responds with the HTTP status code "*403 Forbidden*" to signal that the external system is not allowed to access the requested resource. If, on the other hand, the token is valid, the server grants access to the requested resource.

6 Implementation

In the following sections, specific concepts, technologies as well as implementations are presented that can be used when implementing the rule exchange infrastructure like specified in chapter 5. More precisely, this means that for each of the three components technologies and concepts were found that allow an implementation according to the specification and by following the requirements.

6.1 Server component

Because of its platform independence, the programming language Java is used to implement the server component. To built the REST architecture, the Spring Framework should be used [Sof21]. More precisely, the sub-project Spring Boot should be used as it allows the development of Spring applications by following the software design paradigm *convention over configuration* [Sof]. This reduces the complexity of configurations and forces the developer to follow the optimized design for a REST architecture. In addition to that, Plain Old Java Objects (POJOs) containing Java annotations are used during implementation, which makes the code less complex, easier to understand and more readable. As persistent data storage, the ORDBMS PostgreSQL should be preferred.

6.1.1 Spring architecture

The architecture of a Spring Boot application follows a simple principle. A REST controller contains the endpoints for the REST API. To process requests, each REST controller uses one or more service classes, which contain the actual application logic. The services in turn use repository interfaces for database interaction. Thereby, Dependency Injection is used to instantiate objects a class depends on. For example, a service class gets injected into a controller class by Spring Boot using Dependency Injection. Spring Boot also offers the possibility to perform database updates in a transactional manner. Transactions have the advantage that updates happen according to the *All-or-Nothing* principle and thus the database is always in a consistent state.

An example for the Spring Boot architecture the server component of the rule exchange infrastructure should have is provided in figure 6.1. The rule controller, rule service and rule repository classes are responsible for CRUD operations on rules. As the names suggest, the user controller, user service and user repository classes are responsible for CRUD operations on user accounts and the account controller and account service classes are used whenever a user wants to edit his own account information. There's no need to have an account repository, since the account service also performs CRUD operations on user accounts and thus can simply use the user repository.

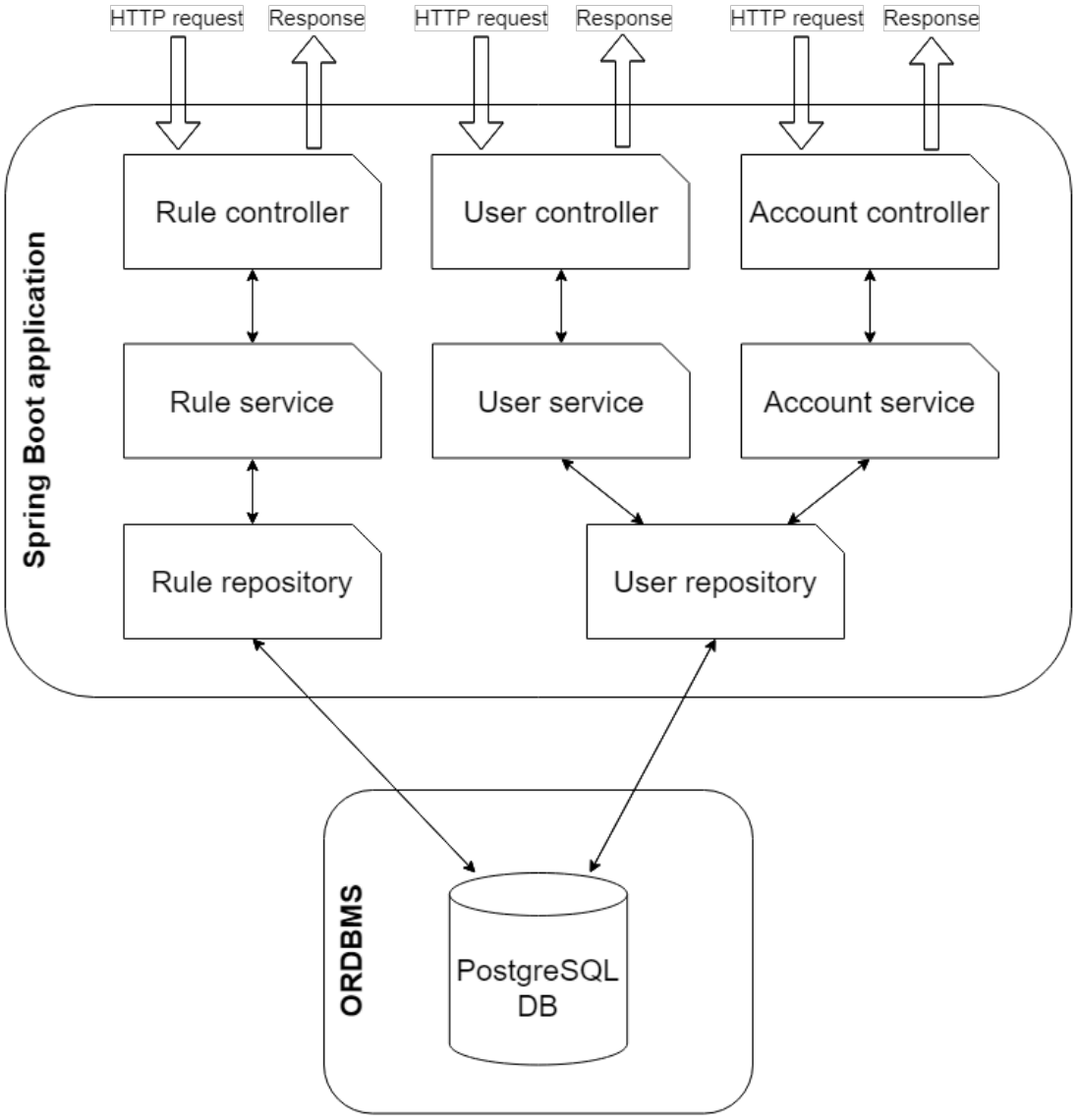

Figure 6.1: Spring Boot architecture of server component

For the authentication and authorization processes described in section 5.5.2 and 5.5.3, in Spring so-called Filters are used. In simplified terms, Spring Filters are POJOs that contain methods which are called before an incoming request can access the actual REST endpoints. There's a need for two filters, an authentication and an authorization filter.

The authentication filter becomes active whenever a POST request is sent to the "*/login*" endpoint. The username and password fields have to be extracted from the body of the request and checked against the user accounts stored in the database. If the credentials match those from a user account, a JWT has to be issued and added to the response like explained in section 5.5.2. If not, the requesting client should receive an informative error response.

Listing 6.1 Example of a REST endpoint inside the rule controller class

```
@RestController
@RequestMapping("rules")
public class RuleController {

  @Autowired
  private IRuleService ruleService;

  @GetMapping("/{id}")
  @ResponseBody
  public ResponseEntity getRuleById(@PathVariable UUID id) {
    Rule rule = this.ruleService.findById(id);
    if (rule != null) {
      return ResponseEntityBuilder.ok(rule);
    } else {
      return ResponseEntityBuilder.notFound("Rule does not exist");
    }
  }
}
```

If an external system tries to access an endpoint which requires authorization, the authorization filter becomes active first. This filter works according to the authorization process described in section 5.5.3. The token is extracted from the HTTP header and an attempt is made to validate it. If the token is valid and non-expired, the role of the user is checked and if it matches the role required to use the endpoint, access to it is granted. Otherwise, if the token is invalid or the role does not match, the request is denied and thus the requesting client is unable to access the endpoint.

To decide whether an endpoint requires authorization or a specific role to be accessed and to add the filters, a security configuration class can be used. This class should also configure the internal password encoder to use the cryptological hash function *bcrypt* to hash user account passwords before storing them and for security reasons, the HTTP methods that can be used to make requests should be restricted to GET, PUT, POST and DELETE. The possibility to outsource the authentication and authorization process to filters and the whole security configuration to a single class is another big advantage of the Spring framework. Spring supports this principle known as Aspect-oriented Programming (AOP) very well. It is also applied when dealing with transactions as previously described. Because of the isolation of all of these aspects, the code from the application logic is clean, easy to understand and readable.

To provide an example of a REST controller containing an endpoint, listing 6.1 shows an endpoint inside a RuleController class, with which clients could download a specific rule by providing its UUID. The *RestController* annotation of the RuleController class tells Spring that this class represents a REST controller and that data returned by a method should be written to the response body, while the *@RequestMapping* annotation specifies that every endpoint in this class has the prefix "*/rules*". As an attribute, the class contains an instance of the rule service interface annotated with *Autowired*. This means that Spring has to instantiate the rule service using the Dependency

Listing 6.2 Example of a method to retrieve a rule inside the rule service

```
@Service
public class RuleService implements IRuleService {

  @Autowired
  private RuleRepository repository;

  public Rule findById(UUID id) {
    return repository.findById(id).orElse(null);
  }
}
```

Injection mechanism. The *GetMapping* annotation specifies that the method *getRuleById()* below it should be called whenever a GET request is made to the route "*/rules/{id}*". The *ResponseBody* annotation is responsible for the serialization of the returned objects into JSON.

The method *getRuleById()* has one parameter of the type UUID annotated with *PathVariable*, so that the UUID of the rule is automatically inserted from the URL into the id parameter. In the body of the method, the first thing that happens is that the *findById()* method of the rule service shown in listing 6.2 is used to retrieve the rule with the given UUID. The rule service either returns the value *null* if there is no rule with the given UUID in the database or the rule object that was loaded from the database using the rule repository. If the rule could be found, the *rule* variable is unequal to null and thus packed into a response entity with the status code "*200 (OK)*" which is then sent back to the client. However, if the rule could not be found, a response entity containing an appropriate error message and the status code "*404 (Not Found)*" is returned.

6.2 Client component

As already mentioned in the specification, the client component must be designed as library used by SAST tools and implemented in the Java programming language. For interaction with the rule exchange server, the best thing would be to use the Spring REST client *RestTemplate*. It allows to easily send HTTP requests to the server and returns the response that was received as *ResponseEntity* object.

6.2.1 Architecture

The client should contain a class containing all the interfaces which are offered to the SAST tool. There, methods to download rules from rule providers as well as methods allowing a simple interaction with the local H2 database have to be provided. The method for downloading all rules has to support the HATEOAS principle and must be able to handle the response from the server, which is in JSON HAL format. Furthermore, the client should have a class containing methods to transform a JSON String into a Java list object that contains the rule objects from the JSON string.

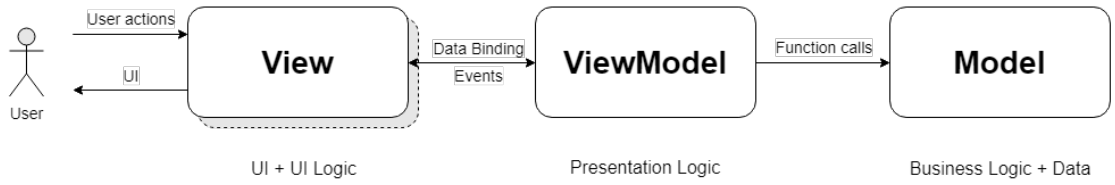

Figure 6.2: Concept of MVVM design pattern

The rest of the implementation, which should not be used directly by the SAST tool, should be located in an *internal* package, to state that it is not intended to be used by programs outside of the library. There, for example the model of a rule and a class handling the actual database interaction using SQL statements should be.

6.3 Administration component

To implement the web application for the administration component, the Angular framework developed by Google LLC should be used [LLC21]. Angular provides some major advantages, including that it uses a Dependency Injection design pattern and it follows a Model View ViewModel (MVVM) architecture, which is a variant of the Model View Controller (MVC) design pattern, allowing to isolate the app logic from the User Interface (UI) layer. In addition to that, there are libraries created by Google LLC for its Material Design making it easier to use it for the web application. In Angular, as programming language TypeScript developed by the Microsoft Corporation is used [Mic]. TypeScript builds on JavaScript, but it allows the use of static type definitions. As the JavaScript runtime Node.js would be the best choice here, as it is a very resource-efficient cross-platform architecture and therefore ideally suited for web applications [Foub].

6.3.1 Architecture

As already mentioned, the architecture of an Angular web application follows the MVVM design pattern like shown in figure 6.2. The *View* layer consists of Hypertext Markup Language (HTML) documents as well as of Cascading Style Sheets (CSS) files and contains all GUI elements that should be shown to the user and also some UI logic. Dynamic content (for example rules) is loaded by binding to the *ViewModel*, which contains the actual UI logic. The *ViewModel* receives events, like for example a button click, triggered by the user from the *View* and handles them. For data loading and event handling, it uses the models, methods and services from the *Model* layer. The *Model* contains the whole business logic and is also responsible for data access. In Angular, both *ViewModel* and *Model* are implemented in the TypeScript programming language.

When using Angular, functions to exchange data with the server are implemented in service classes, which are injected in other services or classes from the *ViewModel* using Dependency Injection. In combination with models for objects (that could represent rules or user accounts) and objects sent or received by the server, they form the Model layer of the application.

Listing 6.3 Example for a rule service

```
    export class RuleService {

      constructor(
      private http: HttpClient,
      private authService: AuthService
      ) {}

      public addRule(rule: Rule): Observable<any> | null {
        var token = this.authService.getToken();
        if (!token) {
          return null;
        }

        let header = new HttpHeaders({ Authorization: 'Bearer ' + token });
        return this.http.put(environment.baseUrl + '/rules/create', rule, { headers: header })
;
      }

      public getAllRules(): Observable<any> {
        return this.http.get(environment.baseUrl + '/rules');
      }

      public getRule(id: string): Observable<Rule> | null {
        return this.http.get<Rule>(environment.baseUrl + '/rules/' + id);
      }
    }
```

To provide an example for a service, a rule service class containing some functions to perform CRUD operations on rules using endpoints of the server defined in section 5.1.2 is shown in listing 6.3. In the constructor an HTTP client object and the authentication service is injected. The HTTP client is used to send requests to the server and the authentication service is responsible to handle the authentication as well as to store and load the JWT required for authorization. The *addRule()* method can be used to create new rules and loads the JWT first. If the JWT is valid and not expired, it is placed in an *HttpHeader* object. After that, a PUT request is created and returned with the rule object of the rule to create in the body and the header object containing the token in the header.

The *getAllRules()* method can be used to download all rules from the server while the *getRule()* method can be used to download a single rule with the given UUID from the server. These two methods do not require authorization and thus also no JWT has to be added to the request.

The rest of Angular web applications is usually split into components. A component basically consists of three parts: A TypeScript class that represents the ViewModel layer and contains the presentation logic of a component as well as an HTML file and a CSS file which together represent the View layer and define the UI and also contain UI logic. This division into components makes the application very flexible, maintainable as well as expandable. It also ensures that a component and especially the code of it does not become too complex. The following components are required to satisfy the requirements and build the user interface specified in section 5.3.1:

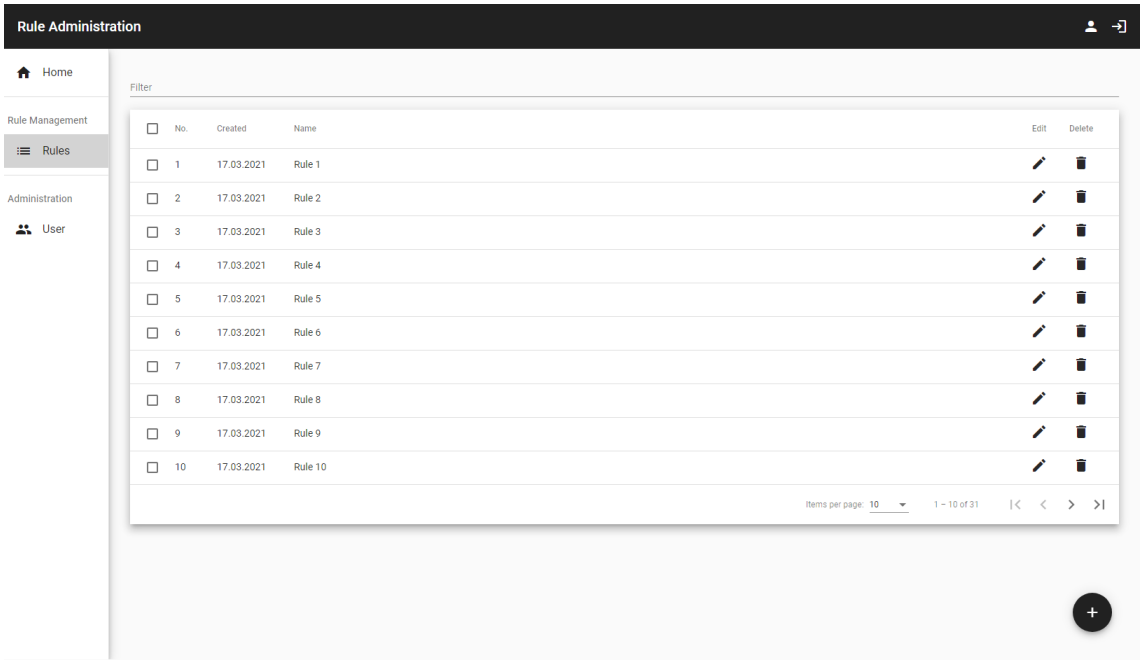

Figure 6.3: Example for the rule page

- A rule component responsible for the rule page. An example for the appearance of it is shown in figure 6.3.
- A user component responsible for the user accounts page that can be accessed by administrators only. For consistency, its appearance should not differ too much from the one the rule page has.
- An account component to handle the editing of personal account information. An example for the appearance of this component is shown in figure 6.5.
- A home component to manage the home page of the application. It acts as landing page after the login and welcomes the user.
- A login component to handle the login page. Like previously specified, it should display a card containing the login form in which a user can enter the login credentials.
- A toolbar component that is responsible for the design and layout of the toolbar.
- A sidenav component that is responsible for the design and layout of the sidenav.
- A footer component that creates a footer. Actually there's no need for a footer, but the opportunity to have one should not be denied for rule providers.
- Dialog components for important dialogs to show to the user.
- An app component, which acts like a parent component and displays the toolbar as well as the sidenav.
- A page-not-found component that is used when an unknown URL is entered. It should display a message that informs the user of an incorrectly entered URL.

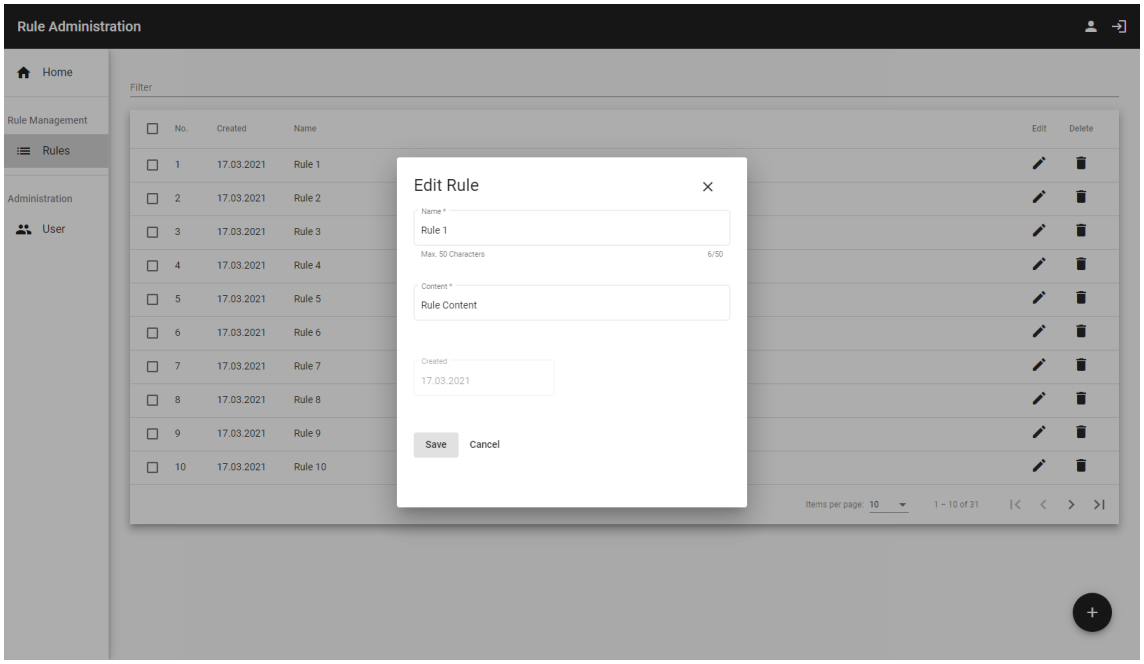

Figure 6.4: Example for the rule page with an open rule editing dialog

6.3.2 User interface

The user interface of the web application like described in section 5.3.1 should mainly use elements from Material Design. In the following, the most important elements for rules and the personal account information are presented and explained. In addition to that, some examples for the user interface were created and are also provided here. To follow the golden user interface design rules, especially the one regarding consistency, the user account administration page should have the same layout as the rule page. Therefore it would be superfluous to also present the user account administration page here.

Figure 6.3 shows the page where the CRUD operations for the rules should be performed. The main element on this page is the table containing all of the rules from the rule provider. In each row, a checkbox to select multiple rules, the row number, the date the rule was created, the name of the rule and two buttons are displayed. The first button should be used to edit the rule while the second button deletes the rule. When clicking the edit button, a dialog like the one shown in figure 6.4 should open allowing the user to edit the rule. To make it difficult to accidentally delete a rule, a confirmation dialog should exist and has to be shown after clicking the delete button. So, the user has to confirm the decision to delete the rule. To create a new rule, the Floating Action Button (FAB) in the right corner at the bottom of the page can be used. When clicking it, a dialog like the one shown in figure 6.4 should open, but with empty fields.

Rules can be searched using the filter which is located above the table. By entering text, the rules in the table below are automatically filtered by their name and content according to the entered text. By clicking on the *Created* or *Name* field in the header of the table, the rules can be sorted ascending or descending according to either the creation date or the name. As the number of rules increases, pagination becomes more and more important. Therefore the table supports this mechanism, which

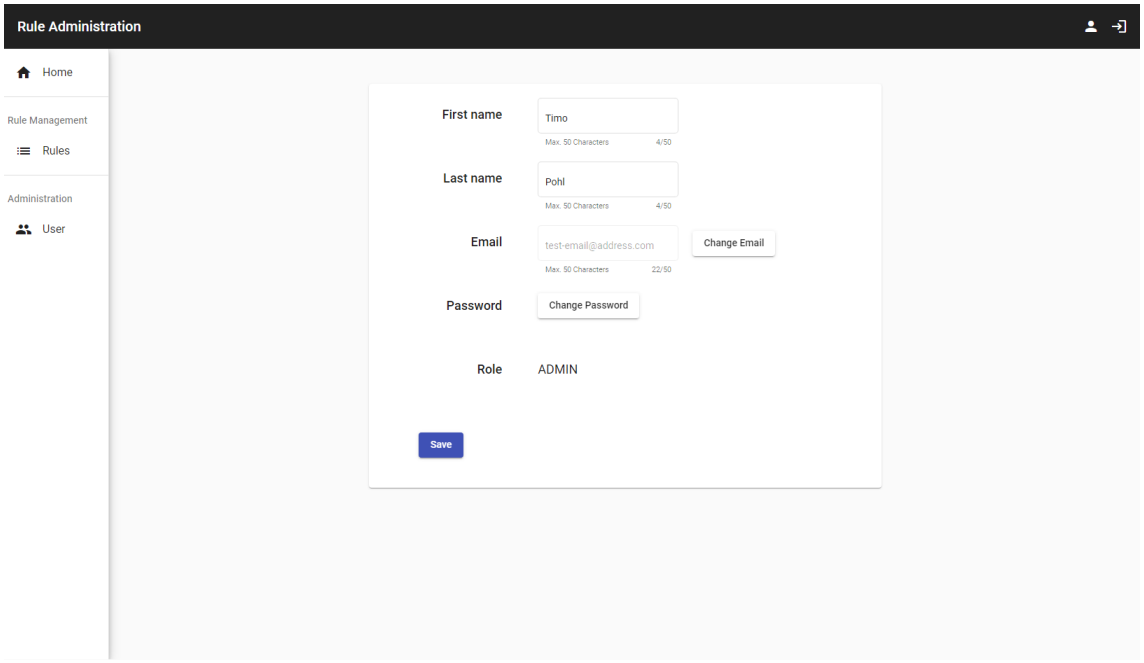

Figure 6.5: Example for the account settings page

can be easily adapted to personal preferences by configuring the items per page at the bottom of the table. With the buttons next to this configuration field, tt is possible to switch pages or jump to the first or last page.

Last but not least, figure 6.5 provides an example for the account settings page, where the personal user information can be edited. When accessing this page, the input field to change the email address is disabled and has to be activated first by clicking the button next to it. This is a precautionary measure and serves to prevent accidental changes, as this could mean that the user can no longer log in. When a user is done editing his account information, the *Save* button can be used to save them. However, if a user wants to change the personal password, the *Change Password* button can be used to open a dialog. There, they have to specify the current password and twice the new password, since this confirms the identity and helps to avoid mistakes.

7 Conclusion

In this work, a data exchange infrastructure for the exchange of rules used by static code analysis tools was created. It enables rule providers to easily distribute their rules and SAST tools to download new sets of rules or update their existing ones at any time. First, the goal was defined and the related work of the university of stuttgart and RIGS IT GmbH was presented. After that, some important basics including standards and the definition of data security were discussed. Following this, the functional and non-functional requirements on which the work is based were shown and explained. Therefore, the infrastructure was divided into a server component, a client component as well as an administration component. Then, the actual specification, which is based on the previously defined concepts and requirements, was explained in detail. To do this, each component was examined and specified one after the other. In the penultimate chapter, specific concepts and technologies that should be used for implementation of each component were presented and explained in detail. These were selected based on the requirements and the specification of the data exchange infrastructure. During the whole process, data security was of utmost importance. However, care was also taken that requirements like maintainability, flexibility and expandability are not neglected.

Outlook

The next logical step is to implement and publish each of the three components of the rule exchange infrastructure. Therefore, the server and administration components together should be made available for rule providers and the client component should be made available for creators of SAST tools. Because of the flexibility of the administration component, each rule provider company is then able to adjust the design of the web application according to their corporate identity and corporate design.

In our digitized world technology evolves very quickly, especially when it comes to quantum computing. With quantum computers, it could one day be possible to bypass cryptographic processes that are considered secure today. Nowadays, there are already quantum security analyzes for cryptographic algorithms such as AES used to encrypt exchanged data in the rule exchange infrastructure [BNS19]. It is believed that the security of the AES algorithm with a 256 bit key can be reduced to about the security of AES with a 128 bit key by using quantum computers in combination with *Grover's algorithm* [GLRS15]. Although this can still be considered secure, it may no longer be the case in a few years, when there might be new cryptanalytic methods or even working quantum computers [Inf20a]. Because of these reasons, official bodies have limited the validity of their statements and guidelines, for example up to the end of the year 2025 in the case of the BSI [Inf20a]. So, the development of this process should always be kept in mind and new recommendations should be responded to immediately. Without such drastic and unpredictable changes in the field of cryptography or quantum computing, the cryptographic methods used here

will be valid until at least 2026 [Inf20b]. However, ideally by the year 2027 at the latest, the security concepts of the rule exchange infrastructure should be re-evaluated on the basis of the state of the art and, if necessary, adjusted.

Bibliography

[BNS19] X. Bonnetain, M. Naya-Plasencia, A. Schrottenloher. "Quantum Security Analysis of AES". In: *IACR Transactions on Symmetric Cryptology* 2019.2 (June 2019), pp. 55–93. DOI: 10.13154/tosc.v2019.i2.55-93. URL: https://hal.inria.fr/hal-02397049 (cit. on p. 51).

[CR20] G. Charest, M. Rogers. *Data Exchange Methods and Considerations*. 2020. URL: https://enterprisearchitecture.harvard.edu/files/enterprise/files/data_exchange_advisory_v1_final.pdf?m=1581437469 (cit. on p. 27).

[Fie00] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000. Chap. 5. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (cit. on pp. 28, 29).

[Fie08] R. T. Fielding. *REST APIs must be hypertext-driven*. 2008. URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (cit. on p. 28).

[Foua] G. Foundation. *GraphQL - A query language for your API*. URL: https://graphql.org/ (cit. on p. 28).

[Foub] O. Foundation. *Node.js website*. URL: https://nodejs.org/en/ (cit. on p. 45).

[GLRS15] M. Grassl, B. Langenberg, M. Roetteler, R. Steinwandt. *Applying Grover's algorithm to AES: quantum resource estimates*. 2015. arXiv: 1512.04965 [quant-ph] (cit. on p. 51).

[ICR19] FBI. *Internet Crime Complaint Center 2019 Internet Crime Report*. 2019. URL: https://pdf.ic3.gov/2019_IC3Report.pdf (cit. on p. 13).

[Inf20a] F. O. for Information Security. *Cryptographic Mechanisms: Recommenations and Key Lengths*. 2020. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.html (cit. on pp. 17, 51).

[Inf20b] F. O. for Information Security. *Cryptographic Mechanisms: Recommenations and Key Lengths: Use of Transport Layer Security (TLS)*. 2020. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.html (cit. on pp. 17, 37, 38, 52).

[JBS15] M. Jones, J. Bradley, N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. IETF, May 2015. URL: https://tools.ietf.org/html/rfc7519 (cit. on p. 40).

[JSMB13] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613 (cit. on p. 15).

[LLC21] G. LLC. *Angular website*. 2016-2021. URL: https://angular.io/ (cit. on p. 45).

[MC19] K. McKay, D. Cooper. *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*. 2019. DOI: 10.6028/NIST.SP.800-52r2. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r2.pdf (cit. on pp. 17, 38).

[Mic] Microsoft. *TypeScript website*. URL: https://www.typescriptlang.org/ (cit. on p. 45).

[ML07] N. Mitra, Y. Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. W3C Recommendation. https://www.w3.org/TR/soap12-part0/. W3C, Apr. 2007 (cit. on p. 28).

[NKMB16] S. Nadi, S. Krüger, M. Mezini, E. Bodden. ""Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs?" In: *38th IEEE International Conference on Software Engineering* (2016) (cit. on p. 13).

[RDM08] A. Roy, A. Datta, J. Mitchell. "Formal Proofs of Cryptographic Security of Diffie-Hellman-Based Protocols". In: vol. 4912. Mar. 2008, pp. 312–329. DOI: 10.1007/978-3-540-78663-4_21 (cit. on p. 38).

[Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. IETF, Aug. 2018. URL: https://tools.ietf.org/html/rfc8446 (cit. on pp. 37, 38).

[RWWM19] H. Rust, N. Wenzel, S. Wagner, K. Mindermann. *CRITICALMATE Vorhabensbeschreibung*. 2019 (cit. on pp. 15, 16, 19).

[SAE+18] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan. "Lessons from building static analysis tools at Google". In: *Communications of the ACM* 61.4 (2018), pp. 58–66. DOI: 10.1145/3188720 (cit. on p. 15).

[Shn98] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 3rd. USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201694972 (cit. on p. 18).

[Sof] P. Software. *Spring Boot website*. URL: https://spring.io/projects/spring-boot (cit. on p. 41).

[Sof21] P. Software. *Spring website*. 2002-2021. URL: https://spring.io/ (cit. on p. 41).

All links were last followed on April 14, 2021.