Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Design & Implementation of a Fraud Detection System for Autonomous Teams

Abschlussarbeit

zur Erlangung des akademischen Grades:

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Angewandte Informatik*

1. Gutachterin: Prof. Dr. Christin Schmidt
2. Gutachter: MSc. Tobias Dumke

Eingereicht von Louis Andrew [s0570624]

Datum
Last updated on Sun Jul 17 08:29:18 UTC 2022

Abstract

[Summary of the thesis]

Contents

1. Introduction **1**
1.1. Background and Motivation . . . 1
1.2. Goal and Scope . . . 2

2. Fundamentals **3**
2.1. Fraud Detection System . . . 3
2.1.1. Statistical Fraud Detection Methods . . . 3
2.1.2. Rule-based Approach . . . 3
2.2. Hypertext Transfer Protocol (HTTP) . . . 4
2.2.1. HTTP Request . . . 4
2.2.2. HTTP Response . . . 4
2.2.3. Header Fields . . . 4
2.2.4. Body . . . 4

3. Requirement Analysis **5**
3.1. Goal . . . 5
3.2. System Environment . . . 5
3.3. State of the Art . . . 5
3.4. Requirements . . . 6
3.4.1. Use Cases . . . 6
3.4.2. User Stories . . . 7
3.4.3. Software Quality Standard . . . 7

4. Conception and Design **10**
4.1. System Architecture . . . 10
4.2. Software Architecture . . . 11
4.3. Technologies . . . 12
4.3.1. User Interface . . . 13
4.3.2. FDS . . . 14
4.3.3. Message Broker / Notification System . . . 14
4.3.4. Database and Caching Memory . . . 14
4.4. Software Design . . . 14
4.4.1. Features . . . 15
4.4.2. Object Model . . . 19
4.4.3. User Interface . . . 25

5. Implementation **29**
5.1. Technologies and Architecture . . . 29
5.1.1. User Interface . . . 29
5.1.2. FDS . . . 30

5.1.3. RabbitMQ 32
5.1.4. Redis 33
5.1.5. MongoDB 33
5.1.6. Docker Compose 33
5.2. Object Models 34
5.2.1. Validation Rule 34
5.2.2. Validation Result 35
5.3. Features 36
5.3.1. Validation Rules Management 36
5.3.2. Customer Validation on a Registration Event 37
5.3.3. Validation Process 38
5.3.4. Notification on Suspicious Cases 42
5.3.5. Database Connection 44
5.3.6. Validation Real Time Progress 44
5.4. User Interface 45
5.4.1. Navigation 45
5.4.2. Rule Management Form 46
5.4.3. Validation Form 48
5.4.4. Validation Progress 49
5.4.5. Runtime Secrets 51

6. Test **52**
6.1. Unit Test 52
6.1.1. FDS 52
6.1.2. UI 53
6.2. Integration Test 54
6.2.1. FDS 54
6.2.2. UI 55

7. Demonstration and Evaluation **56**
7.1. Demonstration 56
7.2. Evaluation 58

8. Conclusion **61**
8.1. Summary 61
8.2. Outlook 61

Bibliography **62**

9. List of Abbreviations **64**

A. Appendix **I**
A.1. Supplemental Source Codes I
A.2. Supplemental Figures III

List of Figures

3.1. Use case diagram of the fraud detection system 6

4.1. System architecture diagram 10
4.2. Software architecture diagram 11
4.3. Software architecture diagram with technologies used 13
4.4. System sequence diagram for a customer validation when a new customer is registered 15
4.5. System sequence diagram for notifications on suspicious cases 16
4.6. System sequence diagram for validation rules management 17
4.7. System sequence diagram for validation rules management 19
4.8. UML diagram of the validation rule model 21
4.9. UML diagram of the validation result model 23
4.10. Mock up of the rule management form page 26
4.11. Mock up of the validation form page 27
4.12. Mock up of the validation progress page 28

5.1. Screenshot of the header to navigate the UI 46
5.2. Screenshot of the rule management form 47
5.3. Screenshot of the "Operator" select field options, based on the current "Type" value 47
5.4. Screenshot of the autocomplete input usage 48
5.5. Screenshot of the validation form 49
5.6. Screenshot of the validation progress in real time 50
5.7. Screenshot of a component to list available runtime secrets 51

7.1. Screenshot of a runtime secret creation 57
7.2. Screenshot of the validation list page 58
7.3. Screenshot of an email sent by AMQP consumer when a validation process completed with a fraud score exceeding 0.5 58

A.1. UML diagram of the customer model III
A.2. Flow diagram of a validation process III

List of Tables

3.1. Systems and software quality standard based on ISO 25010 and its importance 7

7.1. Implementations of the primary use cases of the system 58
7.2. Completion Table of the Systems and Software Quality Standard (ISO 25010) 59

Listings

4.1. Validation rule example (JSON) 21
4.2. Validation rule **condition** attribute example with ALL condition (JSON) 22
4.3. Validation rule **condition** attribute example with ANY condition (JSON) 22
4.4. Validation result example (JSON) 24

5.1. Dockerfile for UI (Docker) 30
5.2. Configuring database type and URL (Prisma) 31
5.3. Dockerfile for FDS (Docker) 32
5.4. Running a RabbitMQ instance with Docker (Shell) 32
5.5. Running a Redis instance with Docker (Shell) 33
5.6. Configuring environment variables of FDS (YAML) 34
5.7. TypeScript interface of a validation rule (TypeScript) 35
5.8. TypeScript interface of a validation result (TypeScript) 36
5.9. Establishing database connection with Prisma (TypeScript) 37
5.10. Example usage of Prisma (TypeScript) 37
5.11. HTTP controller to schedule a validation process (TypeScript) 37
5.12. ValidationEngine class builder pattern using method chaining (TypeScript) 39
5.13. Accessing runtime information using JSONPath expression (TypeScript) 39
5.14. Usage of the singleton pattern and dependency injection in Agent class (TypeScript) 40
5.15. NumberOperator example (TypeScript) 41
5.16. The usage of OperatorFactory class in the Evaluator class (TypeScript) . 41
5.17. EvaluatorFactory usage in ValidationEngine class (TypeScript) 42
5.18. Openning a connection to RabbitMQ instance (TypeScript) 42
5.19. Publishing a validation result to the RabbitMQ exchange (TypeScript) . 43
5.20. Consuming a message published to the RabbitMQ exchange (TypeScript) 43
5.21. Publishing events on certain validation events using the EventEmitter (TypeScript) 44
5.22. Writing to SSE stream when certain events are published (TypeScript) . 45
5.23. Prefilling form values with a sample customer data (TypeScript) 48
5.24. Redirecting to the progress page when a validation process is scheduled successfully (TypeScript) 49
5.25. Using the EventSource API in the browser (TypeScript) 49

6.1. Dependency injection usage in a unit test within FDS project (TypeScript) 52
6.2. Example unit test of the condition evaluator (TypeScript) 53
6.3. Example unit test of a UI component (TypeScript) 53
6.4. Setting up a testing server environment for integration test (TypeScript) 54
6.5. Integration testing on the FDS (TypeScript) 54

6.6. Integration test on the UI (TypeScript) 55

A.1. *Prisma* schema of a validation rule (Prisma) I
A.2. Dockerfile to run MongoDB as a replication set locally. Taken from Prisma GitHub repository (Docker) I
A.3. Docker Compose usage (YAML) II

1. Introduction

Fraud is an activity where someone intentionally deceives another person / system for any unlawful gain. The need to prevent as many fraud activities as possible should be one of the main priorities for businesses, as the number of fraud cases increase every year. In year 2021, the US Federal Trade Commission (FTC) received 2.8 million fraud reports, 70% more in comparison to the fraud reports in 2020 [9]. Many businesses might already have some experience in handling such fraud cases, but an automated system that could detect and possibly prevent fraud activity with minimal supervision would be beneficial to reduce future risks while providing the possibility and capacity to scale their product.

Fraud detection system is a system or program that uses a set of processes or techniques to detect fraudulent activities based on the input data in an automated way. A fraud detection system could also possibly prevent further fraud activities by running a certain action (e.g. Blocking a fraudulent customer).

As a business scale, it is often a good idea to split the responsibility of a certain domain to its own team, consisting of several people that focus solely on the given area. Large businesses are often built on top of multiple teams, working together as a whole, but usually handle their own responsibilities, have their own goals and use a different technology stack and practices. Given the architecture principles and the autonomy of the teams, how could fraud detection and prevention centrally be managed?

1.1. Background and Motivation

With multiple teams working simultaneously, it is possible that some security checks already exist. Having this in mind, a common ground to integrate existing and a possibility to implement new security checks should be established. Unfortunately, every team has their own agenda and priorities, making it almost impossible to build a unified system that scales without having a huge inter-team dependency.

A solution would be that every team implements some fraud detection in their systems and provides an interface to access the data to be used by other services. As part of the solution proposed, a centralized system is needed to act as a *gateway* that is easily accessible. The centralized system can provide a possibility to build a pipeline of security checks, enabling the user to create a custom flow of checks out of the existing services. Grouping all the check results in a single list would be convenient for the user in interpreting the check results from various checks across multiple teams as a whole. The interface of the system itself should be usable by anyone without any technical background. Hence, making it even easier to manage fraud activity.

1.2. Goal and Scope

The goal of this research project is to explore the possibility to build a system to detect and/or prevent fraud activities while providing the opportunity for multiple autonomous teams to contribute to the process by leveraging their domain knowledge and expertise.

This project won't necessarily undertake aspects such as GDPR compliance and authentication process. Therefore, the system is not ready for production and further improvements on these aspects are required. The result of this research project is an explorative work, and may be used as a base for future projects in similar domain.

2. Fundamentals

This chapter describes the context of the research project and acts as a bridge to provide the reader a better understanding on necessary aspects before moving forward to the requirement analysis.

2.1. Fraud Detection System

Fraudulent activities have always been a problem for businesses and can even be traced back to the year 300 B.C. [5], when the earliest attempt of a fraud activity is recorded.

By learning from previous mistakes and with help of the rapid progress of technology, fraud detection techniques are developed to prevent further damage done by malicious attempts.

A fraud detection system is a system that incorporates one or more fraud detection technique to detect any fraudulent entity. A fraud detection system works by accepting an input data and returning a sort of identifier that determines whether the entity is fraudulent. Usually, the output is a numerical value that represents the probability of the entity being a fraud.

2.1.1. Statistical Fraud Detection Methods

Nowadays, statistical fraud detection methods are widely used to detect fraudulent entities. There are two types of statistical fraud detection; *supervised* and *unsupervised*. According to [1], a supervised fraud detection method works by training a model to make a clear distinction between a fraudulent and non-fraudulent entity. In comparison, an unsupervised fraud detection method identifies a fraudulent entity by specifying an unusual behavior or attribute of the certain entity.

2.1.2. Rule-based Approach

Rule-based approach fraud detection technique is an unsupervised fraud detection method that evaluates a certain entity against a pre-defined list of rules. Bolton and David mentioned in [1], that an unsupervised method is useful to collect data of how a fraudulent entity might be when there is not much prior knowledge.

Unfortunately, Kou et al. argued in [7], that rule-based approach can be difficult to manage, as the rules require a time-consuming configuration for each fraud possibility as well as the adaptation of the rules itself.

2.2. Hypertext Transfer Protocol (HTTP)

According to the official specification, HTTP is an *"an application-level protocol for distributed, collaborative, hypermedia information systems"* [10]. HTTP is often used to transfer multimedia data between a client and a server in a client-server architecture.

2.2.1. HTTP Request

An HTTP request is a request message, sent by a client to the server, containing information such as the HTTP method to be applied, identifier of the resource as well as the HTTP version to be used [10, "5 Request"]. The main purpose of an HTTP request is to apply a particular method[1] on the resource located on the server.

An HTTP request usually contains an absolute request-URI, which specifies the URI of the resource, on which the request should be applied.

2.2.2. HTTP Response

An HTTP response is the response from the server as the interpretation result of a particular HTTP request [10, 6 Response]. An HTTP response contains a status-line, which includes a status code and status phrase as an identifier on how the server interprets the HTTP request[2].

2.2.3. Header Fields

Header fields are additional information passed in an HTTP message [10, "4.2 Message Headers"]. In an HTTP request message, request headers are additional information sent by the client to the server and acts as a request modifier [10, "5.3 Request Header Fields"]. In an HTTP response message, response headers are additional information from the client to the server, containing information regarding the response out of the corresponding status line [10, "6.2 Response Header Fields"].

2.2.4. Body

A message body is used to transfer an entity body of an HTTP message, and it contains the data transferred by the client (HTTP request message body) or by the server (HTTP response message body) [10, "Message Body"]. The message body is optional, it can only be used if the method specification of the particular request allows it.

[1]The HTTP methods available for an HTTP request can be found in [10, "5.1.1 Method"].

[2]The complete definition on each status-code and its meaning can be found in [10, "10 Status Code Definitions"].

3. Requirement Analysis

In this chapter, the requirements of the system will be analyzed based on the problem statement described in the previous chapter. Moreover, the main goal of the software as well as other additional specifications will be discussed.

The fraud detection technique used for this research project is the rule-based approach. Even though there are some disadvantages of using the rule-based approach, it is the simplest technique to implement and the most suitable to collaborate on.

3.1. Goal

The goal of the research project is to build a fraud detection system that provides a possibility for different teams to contribute on the fraud detection process independently, based on their views and knowledge on the characteristics of a potentially fraudulent customer. In such a way, the system can help in transforming the fraud detection process to be a collaborative process and each team can make a contribution by making it more reliable as their knowledge increases. The system should enable collaboration across autonomous teams in a fraud detection process and simplify it by encapsulating the internal logic while presenting the result in an understandable format. The system is responsible primarily in determining whether a customer is likely fraudulent based on the information contributed by the different teams and notifying the concerned parties on certain cases when needed.

3.2. System Environment

The system is initially designed to be used by companies that consist of multiple, autonomous teams with different background and responsibilities. The system is also suitable for companies with a dedicated security team, as the system could support them by providing a graphical interface of a fraud detection process.

3.3. State of the Art

Fraud detection and prevention intrinsically is a big topic. There are many scientific papers published regarding fraud detection / prevention, suggesting approaches with different methods and algorithms to create a system that could predict a fraud case as accurately as possible.

There are also quite a lot of service providers that offer a tailor-built fraud detection system, usually works using machine learning to analyze the data provided to the system.

However, this research project intends neither to come up with a new technique to better predict a fraud case nor to compete with the existing service providers, but rather to explore a possibility of how the domain knowledge from multiple independent teams can be combined to produce a reliable fraud detection system, enabling collaboration across different domains.

3.4. Requirements

This chapter lists the requirements of the system by extracting the use cases from the goal of the research project and its priorities.

3.4.1. Use Cases

A use case diagram is created to better visualize the flow and possible use cases of the system on a high level based on the goal of the research project.

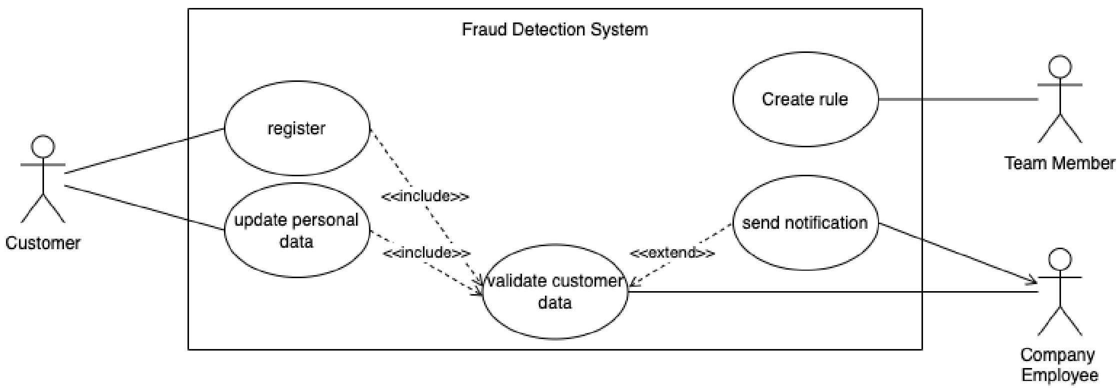

Figure 3.1.: *Use case diagram of the fraud detection system*

The use case diagram gives a visual representation of the main functionality of the system as well as the actor involved in the process. The customer data validation will be run whenever a customer registers or updates his/her personal data. By running a validation whenever an operation on personal data is executed, the system would ideally validate and identify fraudulent customer as soon as possible.

Whenever a fraudulent customer is identified, a notification will be sent to concerned parties, so that necessary actions (for example, blocking the customer) can be taken as soon as possible.

The diagram also visualizes a use case of a validation rule creation by a team member, as a form of contribution based on his/her knowledge of a fraudulent customer with the intention to improve the reliability of the validation process.

3.4.2. User Stories

Based on the use cases visualized on Figure 3.1, the following user stories can be defined:

- **Verifying Customer Early**

  As a stakeholder, I want to verify customers, so that the company can have more confidence that the existing user base is trustworthy

- **Notification on Suspicious Cases**

  As an employee, I want to be notified when a user seems suspicious, so that I can do necessary actions accordingly

- **Validation Rules Management**

  As an employee, I want to manage my own rule to validate users, so that I can use my expertise to find suspicious customers as efficiently as possible without the communication overhead with other teams

The user stories listed above are the main requirements of the system. The functionalities of the system will be defined and implemented based on these user stories.

3.4.3. Software Quality Standard

The system and software quality models defined in ISO 25010 [6] are used to establish a base on how to evaluate the quality of the system made for this research project. The table below contains a list of criteria taken from the software quality characteristics and its sub-characteristics described in ISO 20510, the meaning of each sub-characteristic and its importance to the research project. The criteria will be revisited during evaluation to assess the quality of the software made during the research project.

Table 3.1.: *Systems and software quality standard based on ISO 25010 and its importance*

	Overview	Importance
Functional stability		
Completeness	System covers all the specified tasks listed on the requirement analysis	Very important
Correctness	System provides correct results of the tasks listed on the requirement analysis	Very important
Appropriateness	System accomplishes to fulfill the tasks listed on the requirement analysis in a well manner	Important
Reliability		
Maturity	System is stable during every day use	Important
Availability	System is operational and accessible (ideally via a web browser and no installation is needed)	Very important

Fault tolerance	System still operates well enough, despite software fault	Important
Recoverability	System can recover data in the event of an interruption or failure	Very important
Performance efficiency		
Time behavior	Response and processing time of the system is reasonable	Important
Resource Utilization	The amount and types of resources used by the system is kept as minimum as possible	Not important
Capacity	Maximum limits of a system parameter is within a reasonable range for everyday use	Not important
Usability		
Appropriateness Recognizability	It can be easily recognized, that the system is suitable for the current user need	Not important
Learnability	It's easy to learn how to use the system	Not important
Operability	The system is easy to operate	Very important
User Error Protection	System can protect or prevent users against making errors	Somewhat important
User Interface Aesthetics	User interface is aesthetically pleasing	Very important
Accessibility	System can be used with the widest range of characteristics and capabilities	Not important
Security		
Confidentiality	System is able to ensure that data is only accessible to those who have authorized access	Somewhat important
Integrity	System is able to prevent unauthorized access and modification to computer programs	Somewhat important
Non-repudiation	Actions or events can be proven to have taken place	Somewhat important
Accountability	Actions of an unauthorized user can be traced back	Not important
Authenticity	How well the identity of a subject / resource can be proved	Not important
Compatibility		
Co-existence	Each component of the system can work efficiently without any bottleneck while sharing the same environment	Somewhat important
Interoperability	Two or more systems, products, or components are able to exchange information and use it	Very important
Maintainability		
Modularity	Component of system can be changed with minimal impact on the other component	Important
Reusability	Assets can be used in more than one component	Not important

Analysability	Activities within the system can be easily analyzed (e.g.: in form of logging)	Somewhat important
Modifiability	System can be modified without introducing defects or degrading existing product quality	Very important
Testability	Test criteria for a system is effective and preferably can be run automatically	Very important
Portability		
Adaptability	System can be adapted for different or evolving HW, SW or other usage environment	Not important
Installability	System can be un- and/or installed successfully	Important
Replaceability	System as a product can replace another comparable product	Not important

4. Conception and Design

As the requirements are analyzed, the concept and design of the system can now be defined. This chapter describes the structure, functionalities, technologies and patterns used by the system to fulfill the requirements listed.

4.1. System Architecture

A system architecture diagram is created to help to understand the system as well as its components better.

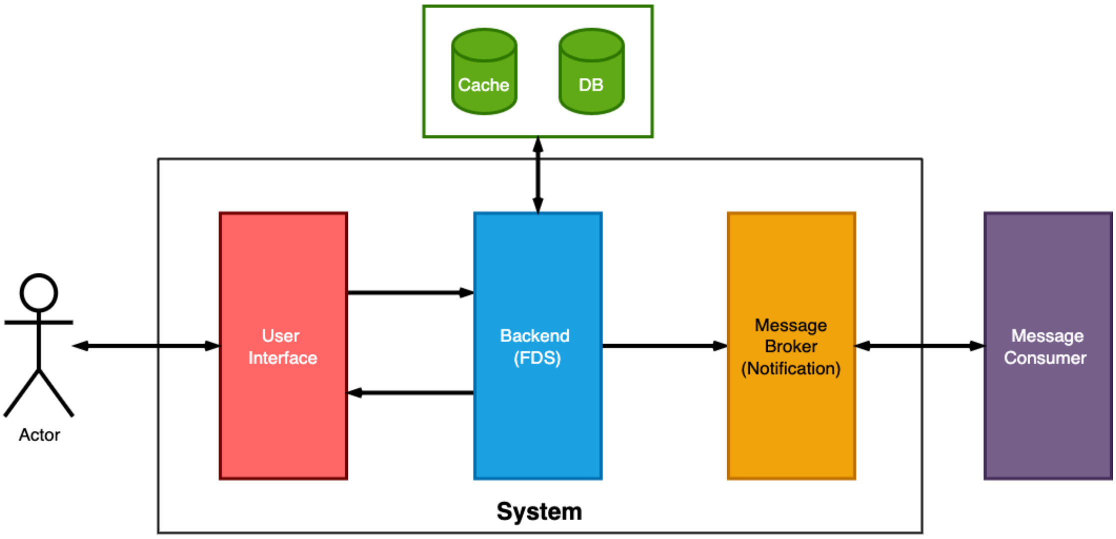

Figure 4.1.: *System architecture diagram*

The system diagram visualizes the components of the system and their interaction with each other. Internally, the system contains 3 independent components; user interface (UI), fraud detection service (FDS) and a message broker (notification system). Externally, the system would interact with a database and optionally a cache memory[1] to persist information needed for the validation process. The diagram also visualizes an external system (*message consumer*) which is indeed out of the system's scope, but plays an important role to determine which action should be taken on certain events. Further information on the function as well as connection between each component will be discussed on the next section.

[1]Cache memories are small, but extremely fast memory used in computer systems to store information that are going to be accessed in a small timeframe [14].

4.2. Software Architecture

As the requirement is clear and the components of the system are defined, a software architecture is needed. A software architecture plays a major role in a software development project by providing a structure on how the software should be built and decisions made during this stage would be vital for the development process going forward. As Garlan wrote in one of his work *Software Architecture*, a software architecture "plays a key role as a bridge between requirements and implementation."[4] A software architecture diagram is therefore created to help visualize the structure, functions and role of each component of the system.

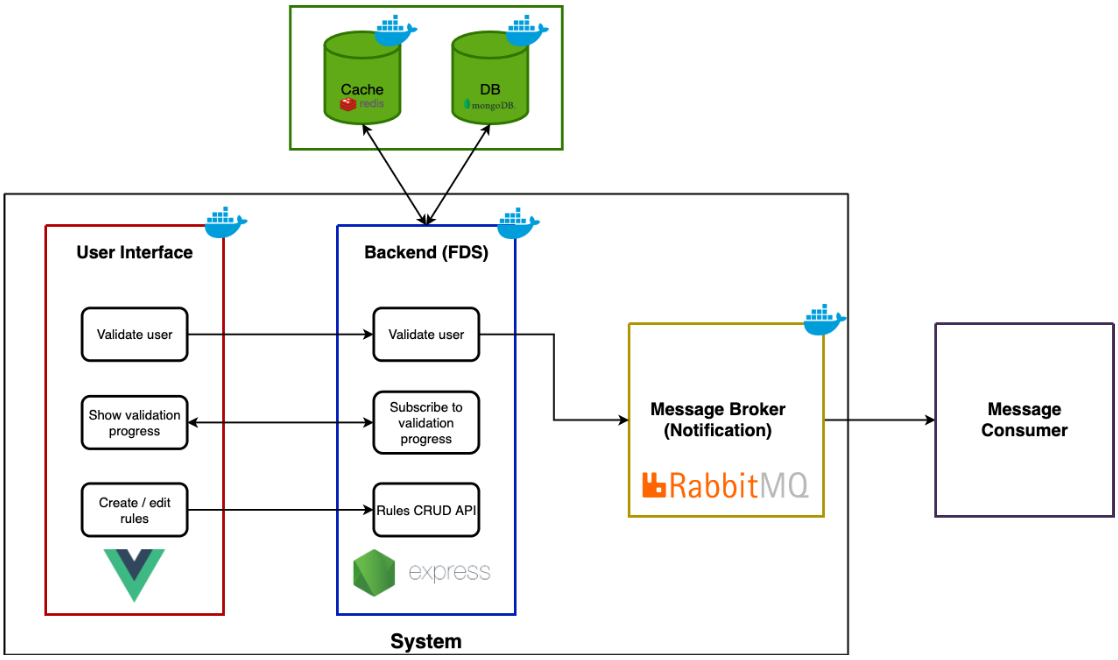

Figure 4.2.: *Software architecture diagram*

In a real world production environment, the user interface might need to be separated into several independent applications. A dedicated app to manage validation rules should be reachable only for internal employees (such as a developer from the company) while a customer facing UI that contains a registration form could also trigger a validation process directly after a new registration. An additional UI to display the progress of the currently running validation processes could also be built specifically for security agents of the companys, so that a fast reaction on certain suspicious customers can be done. For this project, all the use cases mentioned above will be implemented and combined into a single web application.

The fraud detection service (*backend, FDS*) is the core engine of the system where validation processes would be run. The FDS is responsible for all CRUD operations regarding the validation rules. User should be able to create, read, update and delete validation rules via an HTTP request. A detailed explanation on validation rules will be discussed in subchapter 4.4.2. A database connection should be established on the FDS to persist the validation rules. An optional connection to a cache memory could also be established to enable a faster access to the data needed.

The core functionality of the FDS is to run validation process of a customer by evaluating a collection of validation rules in relation to a given customer data. The execution of the validation process could be scheduled via a single HTTP request that contains the customer data on its request body. A validation process is run asynchronously, the FDS will not return the result of the validation directly as a response to the HTTP request. This is intended to prevent a slow response time of the FDS.

Clients could then subscribe to the latest progress of a validation process by accessing an additional endpoint provided by the FDS. A subscription mechanism will be implemented to prevent the need of a request polling on the client side, either by using the WebSocket protocol[2] or something similar.

After a validation process is completed, the FDS should return the result of a validation and further actions should be handled by external services out of the system's scope. This separation of concern is intended to decouple the execution of a validation process and the processing of its result. As there might be several implications on what a validation result might mean, the validation result will be distributed across multiple clients. To achieve this functionality, the *Observer* [3, pp. 293-303] design pattern will be implemented, and a messaging system is needed to act as a bridge between the message producer and its consumers. In this specific architecture, the FDS will act as a message producer, producing a message containing the validation result to the message broker whenever a validation process is done and the external services will act as message consumers, by consuming a message queue created by the message broker and running actions on certain cases independently.

4.3. Technologies

The software architecture determines not only the structure of the software, but it also helps in defining which type of technologies might be needed to build the system as efficiently as possible. Different technologies have their own advantages and disadvantages, and the goal of this phase is not to find the best technology or the best programming language, but rather to find the most suitable set of technologies given the priorities of the project and preferences of the writer. From the software architecture diagram listed on Figure 4.2, the technologies to be used on the following components should be defined:

- User interface (web application)
- FDS (server-side application)
- Message broker / notification system
- Database and caching memory

The previous software architecture diagram is therefore extended with additional logos of the technology used for each component of the system. All internal components

[2] In [8], Fette and Melnikov introduced the WebSocket protocol as a way to establish a two-way communication between a browser-based client and a remote host without relying on opening multiple HTTP connections.

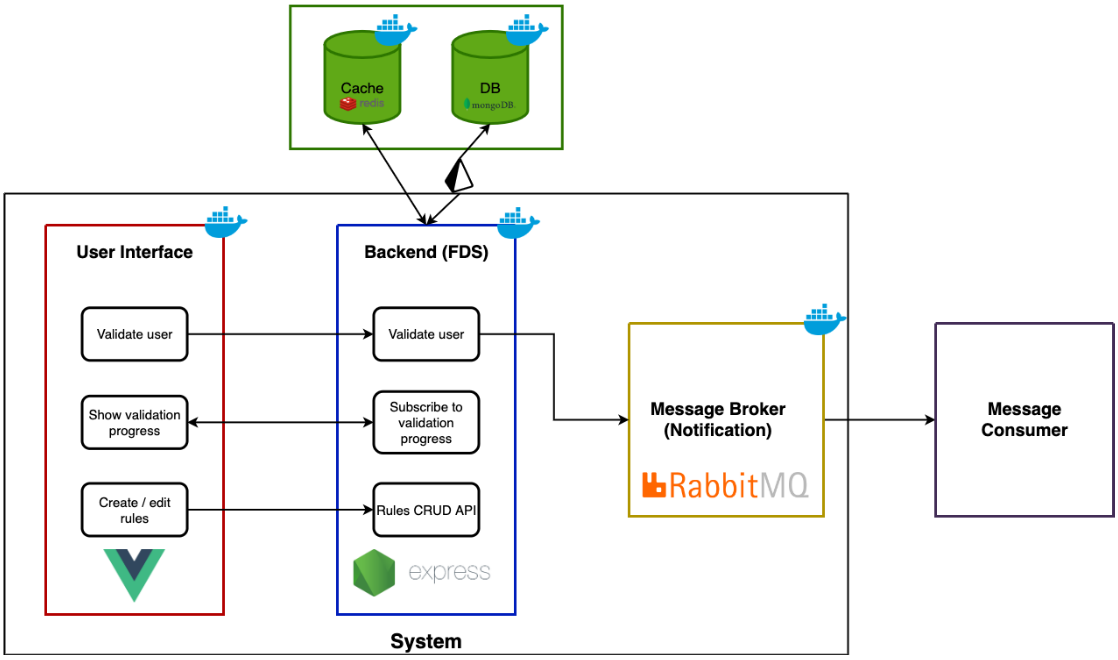

Figure 4.3.: *Software architecture diagram with technologies used*

of the system should be run as a Docker[3] container. Running the applications as a Docker container means that each application is started and run in isolation, ensuring portability to other operating systems. The database and caching memory will also be run as a Docker container, to avoid the need of installing the dependencies needed and to enable the possibility to start all the components of the system using a single command with Docker Compose[4].

4.3.1. User Interface

The technology used to build the user interface is VueJS (3. Version, also known as *Vue3*)[5], a JavaScript framework built on top of HTML, CSS and JavaScript for building a reactive user interfaces using a component-based programming paradigm. To ensure type safety, the user interface is built with TypeScript[6], rather than plain JavaScript, which is also supported by Vue3.

[3] *Docker* is an open source software used to containerize applications in a package with its dependencies and operating system, making it runnable in any environment. Homepage: https://www.docker.com/.

[4] *Docker Compose* is an open source tool for running multiple Docker containers. GitHub repository: `https://github.com/docker/compose`.

[5] *Vue3* is an open source JavaScript framework to build user interfaces. GitHub repository: `https://github.com/vuejs/core`.

[6] *TypeScript* is an open source programming language built by Microsoft on top of JavaScript by adding additional optional static typing. A TypeScript program will be compiled to a plain JavaScript program, before being executed in environment such as browser or NodeJS environment. GitHub repository: `https://github.com/microsoft/TypeScript`.

4.3.2. FDS

The technology chosen to build the FDS is Node.JS[7]. Node.JS is chosen not only because the writer is familiar with it, but also the event loop architecture of Node.JS enables the possibility to perform non-blocking I/O operations asynchronously. Each validation process will be an asynchronous process, which wouldn't block the main thread of the application. To ensure type safety, TypeScript is also used here rather than plain JavaScript. Express.JS[8] is the web framework of choice to build the FDS. Express.JS provides a simple and declarative API to build a web application with ease and speed. An object-relational mapping tool (ORM) is used in this application to provide an easier access to the database, and additionally to keep the database schema in sync between the database server and the FDS application. The ORM of choice for the application is Prisma[9], as it provides a straightforward integration with TypeScript, generating TypeScript types automatically from the database schema.

4.3.3. Message Broker / Notification System

A reliable message broker is needed to make sure that all validation results actually reach every consumer. The technology chosen for this component is RabbitMQ[10], as it is not only reliable, but also has an easy guide to set up as well as a big collection of client libraries for multiple programming languages.

4.3.4. Database and Caching Memory

A database is needed to store data regarding validation rules. Each database system has their own use cases and weaknesses. For this particular project, MongoDB[11] will be used as the database system of choice. Redis[12] is chosen as the technology of choice for the caching memory because of its simple API and reliability.

4.4. Software Design

Software design is an important step that should be done before the implementation of the software. It provides a plan on how the software can fulfill the requirements and decisions made during this process are vital in determining how the software should be implemented. This chapter describes the design process of the system and its features.

[7] *Node.JS* is an open source JavaScript runtime environment that runs on Google's V8 engine, enabling JavaScript programs to be run out of the browser environment. GitHub repository: `https://github.com/nodejs/node`.

[8] *Express JS* is an open source web application framework for Node.JS. GitHub repository: `https://github.com/expressjs/expressjs.com`.

[9] *Prisma* is an open source ORM for Node.JS and TypeScript. GitHub repository: `https://github.com/prisma/prisma`.

[10] RabbitMQ is a messaging broker, enabling the distribution of messages across multiple clients. RabbitMQ homepage: `https://www.rabbitmq.com/`.

[11] *MongoDB* is a source-available NoSQL database developed by MongoDB Inc. MongoDB homepage: `https://www.mongodb.com/`.

[12] *Redis* is an open-source in-memory data structure store. GitHub repository: `https://github.com/redis/redis`.

4.4.1. Features

By analyzing the use cases listed on subchapter 3.4.1, the functions of each component of the system can be defined, and consequentially a sequence of operations will be created for each specific use case. An additional sequence will also be defined as to complete the system and fulfill the requirements defined.

Customer Validation on a Registration Event

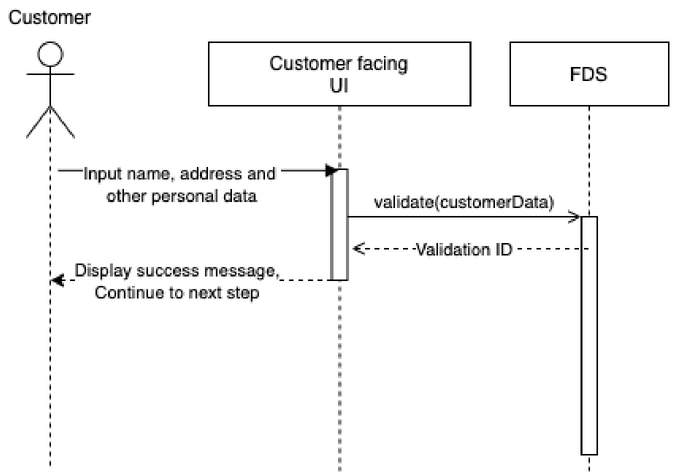

Figure 4.4.: *System sequence diagram for a customer validation when a new customer is registered*

A system sequence can be defined by analyzing the following use case:

> "As a stakeholder, I want to verify customer, so that the company can have more confidence that the existing user base is trustworthy"

One of the opportunity to do a verification process is during a new customer registration. Verifying a customer after each new registration might help the stakeholder to be more confident, that the user base is trustworthy and necessary actions can be taken as soon as possible to reduce the possible damage made in the future by fraudulent customers. The following sequence will be executed as a way to validate a customer directly after a new registration:

- A new customer inputs his or her personal data to a customer facing UI and clicks the *"Register"* button
- The customer facing UI makes an HTTP Post request to the FDS, containing the user's personal data on its request payload
- The FDS receives the HTTP request, and schedules a new validation process to be executed asynchronously
- The FDS responds to the HTTP request by returning a validation ID pointing to the scheduled validation process
- Customer facing UI shows a success message and continues registration to the next step while the validation process runs

Notification on Suspicious Cases

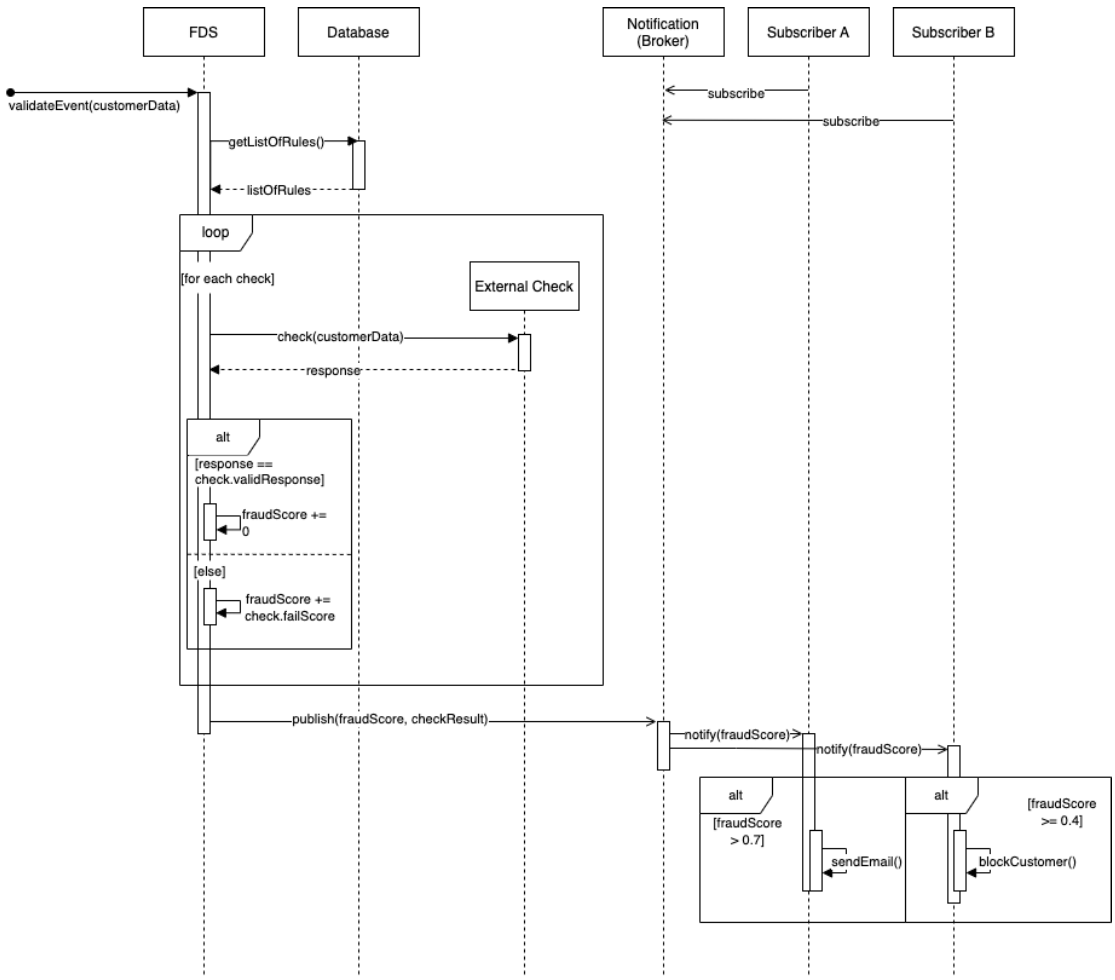

Figure 4.5.: *System sequence diagram for notifications on suspicious cases*

To fulfill the requirements of the system, a further examination of the following use case should be done:

> "As an employee, I want to be notified when a user seems suspicious, so that I can do necessary actions accordingly"

The FDS runs a rule evaluation by making an HTTP request to an external URL and comparing the HTTP response to the conditions listed on the validation rule. As working with external systems is unpredictable and there is no guarantee that the external system has a fast response time, a validation process is run asynchronously, meaning that the FDS would not return the validation result with a resulting fraud score directly to the client when a validation process is scheduled. At the end of a validation process, the concerned parties might need some kind of notification on certain cases, to make sure actions required can be made as soon as possible. The following sequence illustrates the sequence of activities done by the system to validate a certain customer and sending a notification on its completion:

- The FDS receives an HTTP request to schedule a validation process and responds by returning the ID of the validation process
- FDS retrieves a list of validation rules from the database
- FDS begins to initiate a validation process by setting the fraud score to 0 and looping through the list of validation rules for evaluation
- A validation rule will be evaluated by making an HTTP request to the external endpoint defined by the validation rule and evaluating its response according to the condition specified
- If the response matches all the conditions specified by the validation rule, the rule evaluation will be considered as a success and the fraud score will be incremented with 0. Otherwise, the rule evaluation will be considered as a failure and the fraud score will be incremented by the *fail score* specified by the validation rule
- After the evaluation of all validation rules retrieved from the database is completed, the FDS publishes the validation result to an exchange hosted by the message broker
- The message consumers consume the message from the exchange and react accordingly[13]

Managing Validation Rules

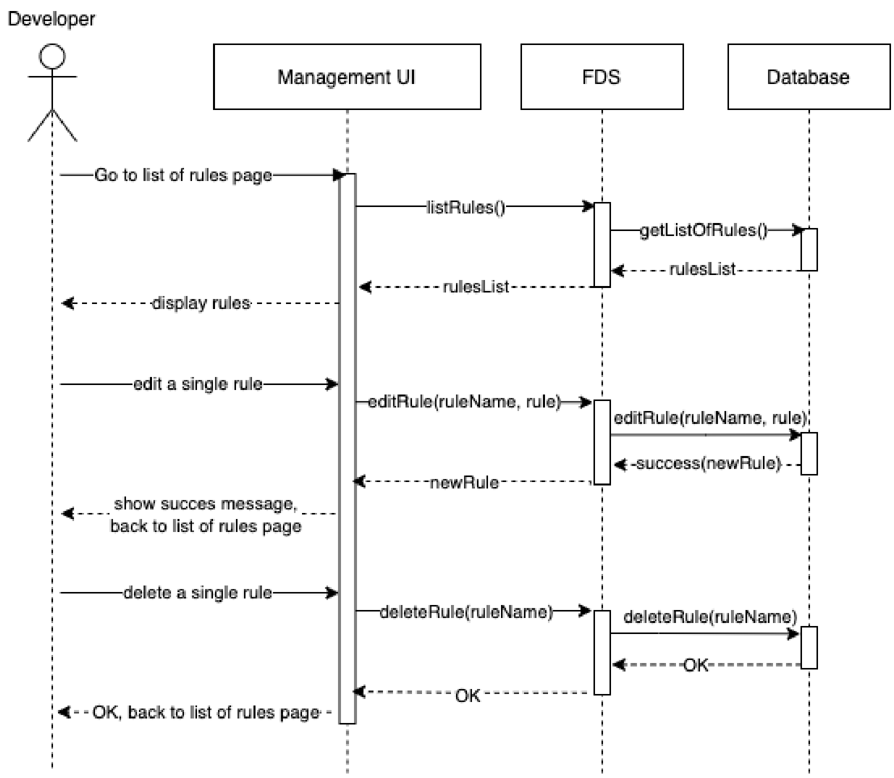

Figure 4.6.: *System sequence diagram for validation rules management*

Another sequence can also be defined as a result of an analysis of the following use

[13]For example: sending an email notification if the fraud score exceeds 0.7.

case:

> "As an employee, I want to manage my own rule to validate users, so that I can use my expertise to find suspicious customers as efficiently as possible without the communication overhead with other teams"

A possibility for each team to manage their own validation rules without being dependent to other teams is needed. By reducing the impediment in the process (having to consult other teams, communication overhead), every team can focus on generating validation rules that reflect a fraudulent customer as efficiently as possible, according to their own domain knowledge and expertise. The following sequence illustrates the functionality of managing validation rules:

- A user (e.g. Developer) can access the management UI and go to the page that displays a list of available validation rules
- The FDS retrieves a list of validation rules from the database
- User can click on a single rule and edit the rule
- The management UI makes an HTTP PUT[14] request to the database to edit an existing rule
- The FDS receives the HTTP request, modifies the rule on the database and returns the edited rule as a response
- The management UI displays a success message and redirects user back to the list of rules page
- User can click on a single rule and delete the rule
- The management UI makes an HTTP DELETE[15] request to the database to delete an existing rule
- The FDS receives the HTTP request and delete the rule on the database, returning a 204[16] status code as an identifier of a successful operation
- The management UI displays a success message and redirects user back to the list of rules page

Validation Real-Time Progress

Even though the user should receive a notification on certain cases, there might be times when a user wants to intentionally monitor the progress of a validation process. To achieve such functionality, the user interface should establish a connection to the FDS, and receive notification whenever there is an update on the validation result. The sequence of such functionality will be as follows:

- The FDS receives an HTTP request to schedule a validation process and responds by returning the ID of the validation process

[14] In [10, "9.6 PUT"], HTTP PUT method is described as a method to store or modify an entity, defined by the Request-URI.

[15] In [10, "9.7 DELETE"], HTTP DELETE method is described as a method to delete a resource on the host server, pointed by the Request-URI.

[16] In [10, "10.2.5 204 No Content"], the 204 status code should be used if the server fulfilled the request, but no data should be returned by the HTTP response.

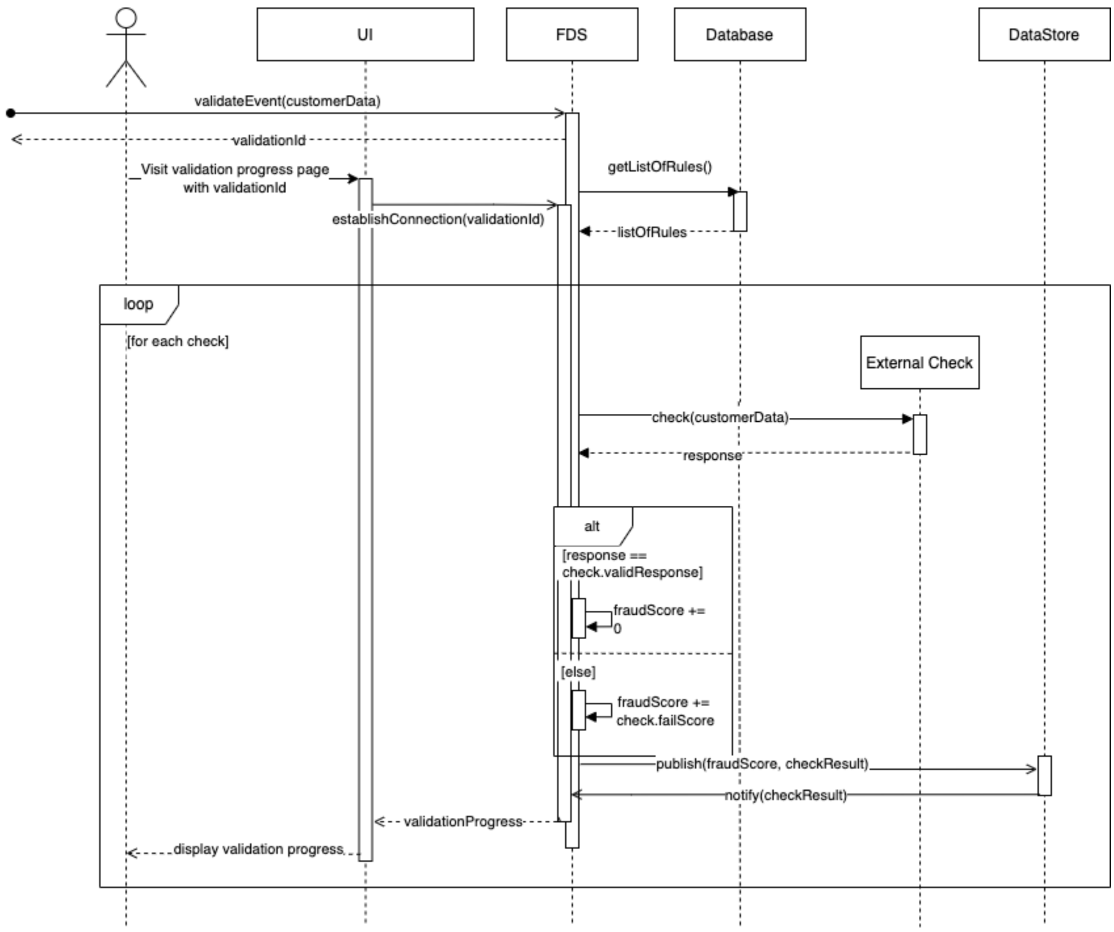

Figure 4.7.: *System sequence diagram for validation rules management*

- FDS retrieves a list of validation rules from the database and initiate the validation process
- A user visits the validation progress page with the returned validation ID
- The user interface establishes a connection with the FDS subscription endpoint
- After each rule evaluation, the FDS stores the latest validation result to a data store[17]
- Every time the data store receives a new data, the user interface will get a notification and the latest result from the FDS subscription endpoint
- The user interface updates the view containing the latest validation result

4.4.2. Object Model

The essential object models of the system can be defined by revisiting the sequences listed on subchapter 4.4.1 and identifying the important structures needed to fulfill the requirements.

[17] A data store in this context can be a caching memory or a simple class to store some temporary information.

Validation Rule

A validation rule is a structure of information used in a validation process by supplying the FDS the necessary information to make an HTTP request to an external endpoint and evaluating its response, affecting the overall fraud score of a validation process through its evaluation result. Through a detailed analysis of the sequence illustrated on Figure 4.5, it is essential that the *validation rule* model contains the following attributes:

- A URL pointing to an external endpoint
- A list of conditions to evaluate the response returned by the external endpoint
- A unique identifier
- A fail score, which determine the severity of the rule if the evaluation failed

It might also be necessary to have an identifier in the validation rule to skip its evaluation in certain cases. Other than that, as the FDS would make an HTTP request to an external endpoint based on the information listed on a validation rule, the following attributes are needed to provide a more robust configuration:

- HTTP method
- Request header fields
- Request body

As the FDS interacts with external endpoints, there is no guarantee that the external endpoint will always be accessible. An additional attribute to specify and configure a retry strategy in such cases can be useful. However, a retry strategy can be really specific to its implementation and therefore will be discussed in section 5.2.1.

An additional `priority` attribute is also provided to enable the possibility to run rule validations according to its priority order.

The `condition` attribute plays an important role for a validation rule, as it defines how the response returned by the external endpoint should be structured to pass a rule evaluation. It is intended to design the condition attribute to be robust and configurable. The `path` of a condition defines a JSONPath[2] expression to access information available of the current validation scope, such as customer information or response returned by the external endpoint. The `type` attribute of a condition determines the data type of the attribute accessed by the `path` attribute. The `type` attribute also determines which operators are available to use[18]. The `operator` attribute refers to a name of operator to be used to evaluate the condition (for example: "eq", "incl"). The available operator names are predefined and restricted to the condition's type attribute. More information regarding operators will be discussed in section 5.3.3. The `failMessage` attribute of a condition refers to a message that is going to be appended to the validation result's `messages` attribute after a validation is completed.

[18] For example: a condition with *type* "string" cannot use the "incl" *operator*, because the "incl" *operator* is only available for "array" *type*.

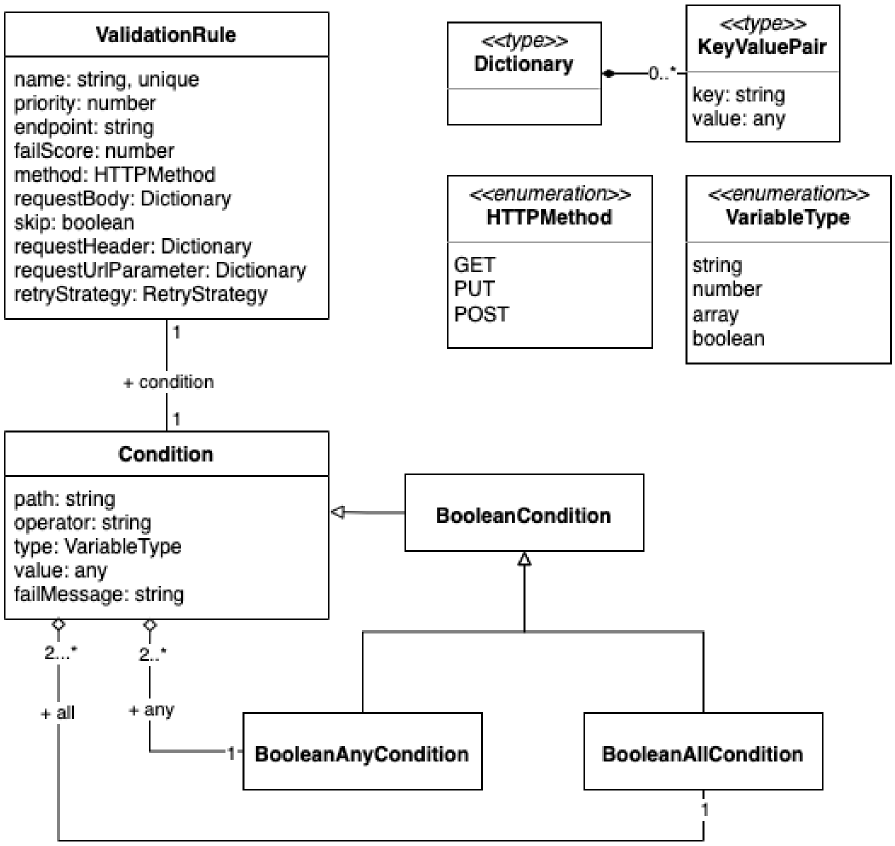

Figure 4.8.: *UML diagram of the validation rule model*

```json
{
  "name": "Example",
  "skip": false,
  "priority": 2,
  "endpoint": "http://localhost:8000/validate",
  "method": "GET",
  "failScore": 0.7,
  "condition": {
    "path": "$.response.statusCode",
    "type": "number",
    "operator": "eq",
    "value": 200,
    "failMessage": "Status code doesn't equal to 200"
  },
  "requestUrlParameter": {},
  "requestBody": {},
  "requestHeader": {}
}
```

Listing 4.1: *Validation rule example (JSON)*

A validation rule might contain more than a single condition to pass an evaluation. In such cases, the user needs to define how to determine whether an evaluation should pass: either pass an evaluation if **ALL** the conditions is true or pass an evaluation if

AT LEAST ONE (ANY) of the conditions is true. This can be achieved by having the `condition` attribute as an object with a single attribute, either *all* or *any* and an array of conditions as the attribute's value.

```json
{
  "condition": {
    "all": [
      {
        "path": "$.response.statusCode",
        "type": "number",
        "operator": "eq",
        "value": 200,
        "failMessage": "Status code doesn't equal to 200"
      },
      {
        "path": "$.response.body.valid_address",
        "type": "boolean",
        "operator": "eq",
        "value": false,
        "failMessage": "Address is invalid"
      }
    ]
  }
}
```

Listing 4.2: *Validation rule* ***condition*** *attribute example with ALL condition (JSON)*

```json
{
  "condition": {
    "any": [
      {
        "path": "$.response.statusCode",
        "type": "number",
        "operator": "eq",
        "value": 200,
        "failMessage": "Status code doesn't equal to 200"
      },
      {
        "path": "$.response.body.valid_address",
        "type": "boolean",
        "operator": "eq",
        "value": false,
        "failMessage": "Address is invalid"
      }
    ]
  }
}
```

Listing 4.3: *Validation rule* ***condition*** *attribute example with ANY condition (JSON)*

Validation Result

A validation result is the result of a validation process. A validation result contains a resulting fraud score, which is the probability of a certain customer being a fraud. The algorithm to calculate the fraud score will be discussed as part of the section 5.3.3. By evaluating the sequences listed on Figure 4.4 and Figure 4.5, the following attributes are needed:

- A unique validation ID
- A fraud score

Furthermore, information regarding the total checks, run checks, and additional information that contains the start and end date of the validation process as well as the customer information used for the validation are essential. The customer information is generic, and the system should be able to do a validation process regardless of its structure. The validation result should also return a list of validation rule names, whose evaluations are skipped due to its `skip` attribute being set to `true`.

Even though the resulting fraud score determines the probability of a customer being a fraud, it is also important to further analyze the actual evaluation result of each validation rule. For example, a certain action can be run if the evaluation of a specific rule failed or the information regarding the rule evaluations can also be used to display the current progress of a validation process (implementation of this specific functionality will be discussed more in subchapter 5.3.6).

To provide such information on a validation result, the `events` attribute will be introduced, which refers to rule evaluation events within a validation process. A rule evaluation event should contain the following attributes:

- Unique identifier of the event (name of the rule being evaluated)
- Status of the event (not started, failed, passed or running)
- If available, start date of a rule evaluation event
- If available, end date of a rule evaluation event
- A list of messages, containing error messages of an evaluation event

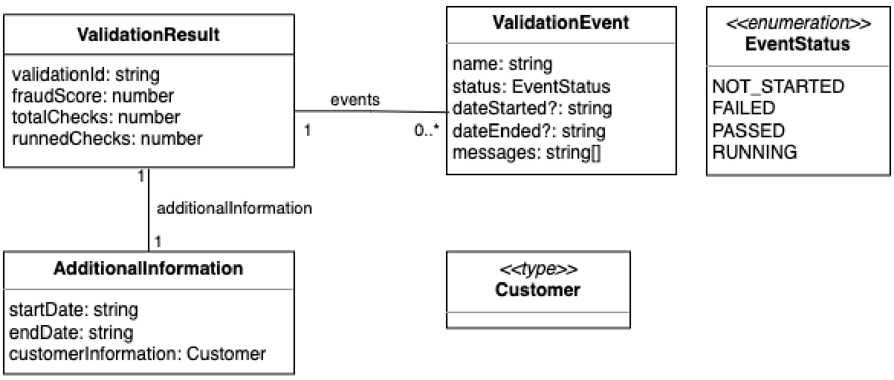

Figure 4.9.: *UML diagram of the validation result model*

```json
{
  "validationId": "3112dc4a-45f5-41b8-883a-715dcbe9a490",
  "totalChecks": 3,
  "runnedChecks": 2,
  "skippedChecks": [
    "Skip rule",
  ],
  "additionalInfo": {
    "startDate": "2022-06-12T09:16:24.618Z",
    "customerInformation": {
      "firstName": "Scooby",
      "lastName": "Doo",
      "address": {
        "streetName": "Suite 5000 185 Berry St",
        "city": "San Fransisco",
        "state": "CA",
        "country": "United States",
        "postalCode": 94107
      },
      "email": "scooby-doo@fraud.co",
      "phoneNumber": "123131123"
    }
  },
  "events": [
    {
      "name": "User's email domain is not blacklisted",
      "status": "PASSED",
      "dateStarted": "2022-06-12T09:16:34.694Z",
      "dateEnded": "2022-06-12T09:16:39.725Z",
      "messages": []
    },
    {
      "name": "User's email is not blacklisted",
      "status": "FAILED",
      "dateStarted": "2022-06-12T09:16:39.725Z",
      "dateEnded": "2022-06-12T09:16:44.756Z",
      "messages": [
        "User's email is blacklisted!"
      ]
    }
  ],
  "fraudScore": 0.425
}
```

Listing 4.4: *Validation result example (JSON)*

4.4.3. User Interface

The UI is the interface for the user to interact with the system. The UI is specifically designed to facilitate the functionalities described in subchapter 4.4.1. Several mock-ups are created to better visualize how the user interface should be structured using a UI design tool called Figma.

Rule Management Form

To facilitate the validation rule management functionality visualized in Figure 4.6, a page containing a form to create, edit, delete and read a validation rule will be created. The rule management form is intended to be used by internal employee, preferably with technical background[19] to manage a validation rule that will be used to validate a customer.

The page should represent every attribute of a validation rule and gives the user the ability to modify the attribute if necessary. The form will be used both for rule creation and rule modification. For a rule creation, the form fields will be left blank. For a rule modification, the form fields will be filled with the rule's current data.

The Rule Name field is used to display or enter a unique name of the validation rule. If the form is used to modify an existing rule, the field should be disabled, as a validation rule's name cannot be modified.

The Conditions section can be used to add one or more condition to a validation rule. The form fields of each condition includes a dedicated input field for each attribute of a condition, as described in section 4.4.2. The Type and Operator form fields are a selectable field, meaning the user has to select one out of several choices provided. This is intended to restrict input from the user, preventing an invalid condition being submitted to the FDS[20]. User can also delete a condition if necessary by clicking the Delete button available on each condition segment. If more than one condition is present, a selectable button will be displayed to select whether the "any" or "all" condition will be used.

As the `retryStrategy` attribute of a validation rule is not required, it is possible to delete an existing retry strategy, by clicking on the Delete button available on the Retry Strategy section of the form. If no retry strategy is present, a button to add a new retry strategy and to display the Limit and Status Codes fields will be displayed.

According to the model of a validation rule, `requestBody`, `requestHeader` and `requestUrlParameter` should be a dictionary that could contain as many entries as possible. To mimic this functionality as a form field, the Request Body, Request Header and Request URL Parameter fields are a dynamic input field. A dynamic input field enables the user to add a new key-value pair to the dictionary by clicking on the Add button and inputting the values to the corresponding input fields. To delete an entry of a dictionary, a delete button is provided next to each key-value field.

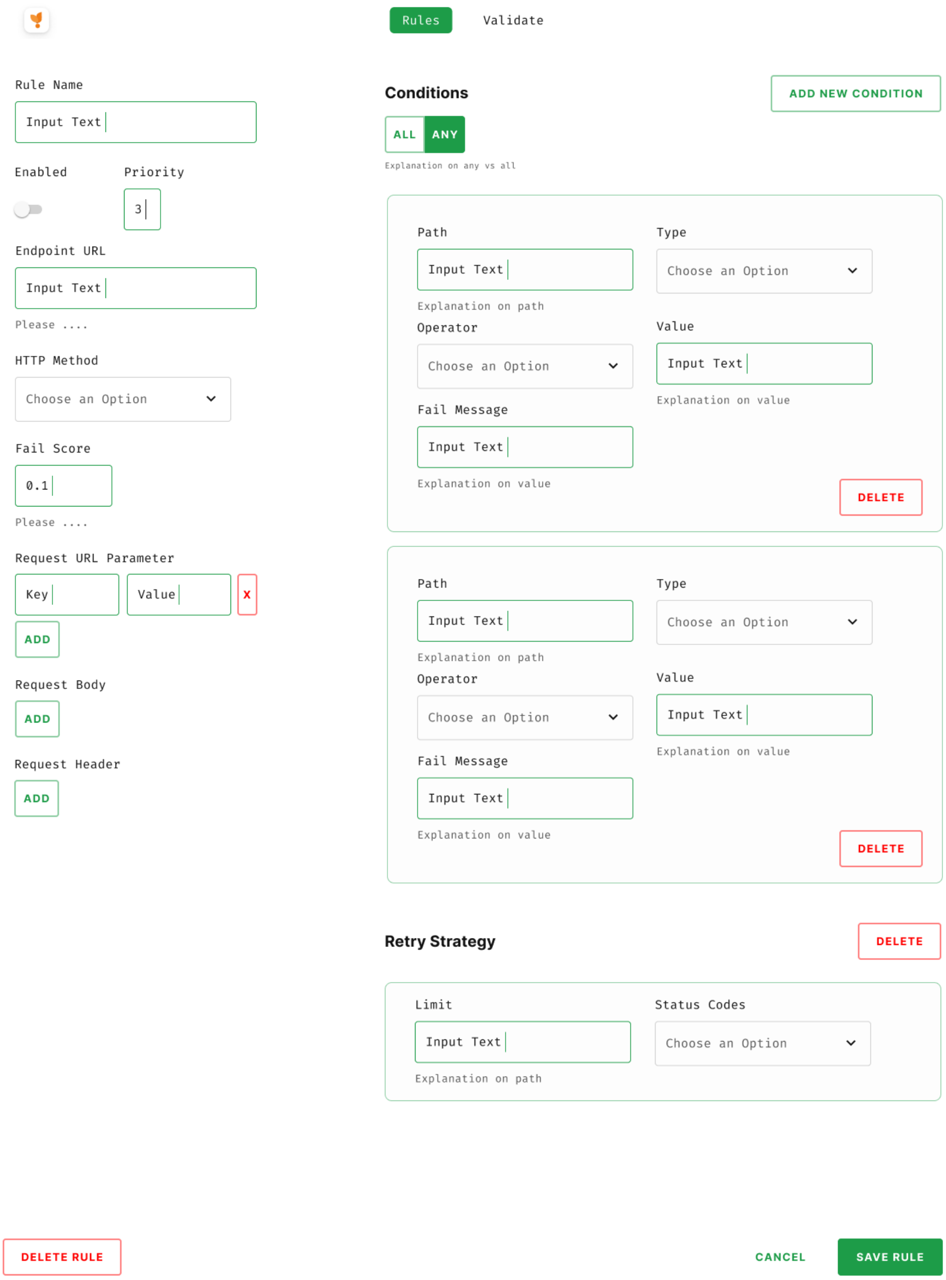

Figure 4.10.: *Mock up of the rule management form page*

Validation Form

A sample registration form is also created to give the user a possibility to run a validation process on a certain customer. The validation form is intended to be used by end customer, as a mean to register his-/herself into the system. Internal employees

[19]An understanding on how HTTP works is a prerequisite to use the form.
[20]For example, on *type* field, user can only choose on of the following: (number, string, array, boolean).

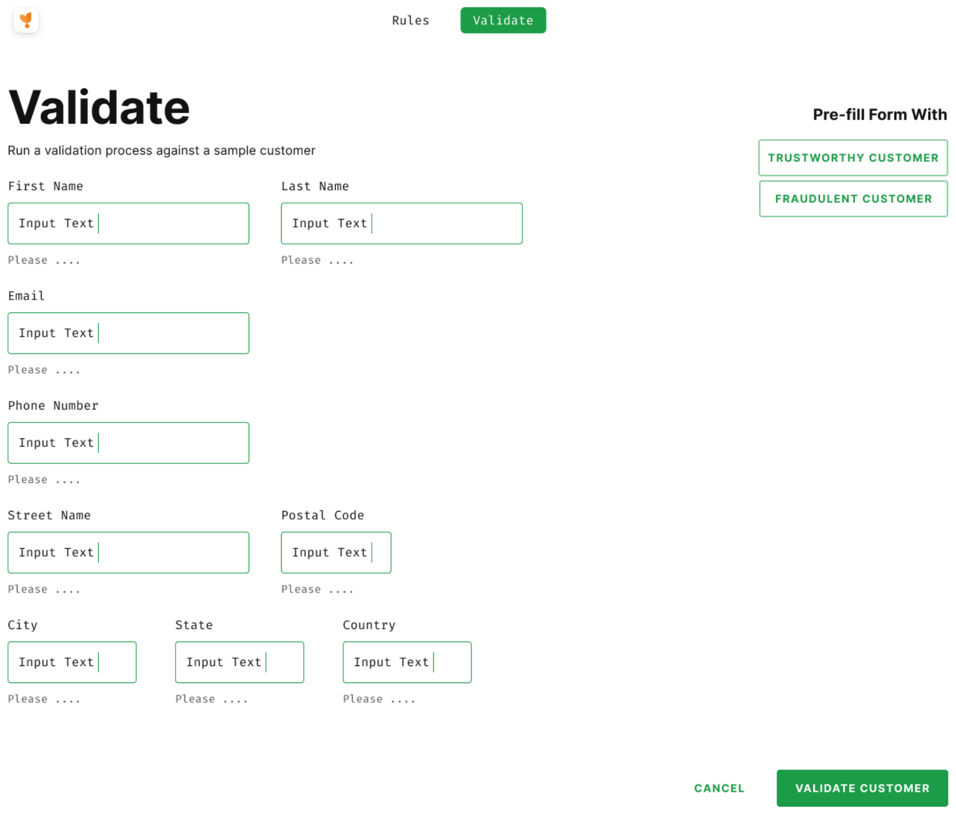

Figure 4.11.: *Mock up of the validation form page*

can also use the validation form to test the validation rules.

The page should represent a customer model by displaying every attribute of a customer in a form field (please refer to Figure A.1 for more information on the customer model). For demonstration purposes, it might be beneficial to have a list of sample customers, so that the user can run a validation on a certain set of customers quickly, without having to first fill out the form by him-/herself. To provide this functionality, a list of buttons, containing a description text of the sample customer will be displayed next to the validation form. Upon clicking on one of the buttons, the validation form will be filled with the sample customer's data and the user can directly click on the VALIDATE CUSTOMER button to begin the validation process.

Validation Progress

To provide a transparency on an asynchronous validation process, a page that displays the current progress of a validation process in real time might be beneficial. This page is intended for demonstration purposes, but can also be beneficial for security agents to keep track of the validation processes run by the FDS.

The page displays the current progress of a certain validation in real time. It represents the `events` attribute of a validation result in a timeline, displaying the events

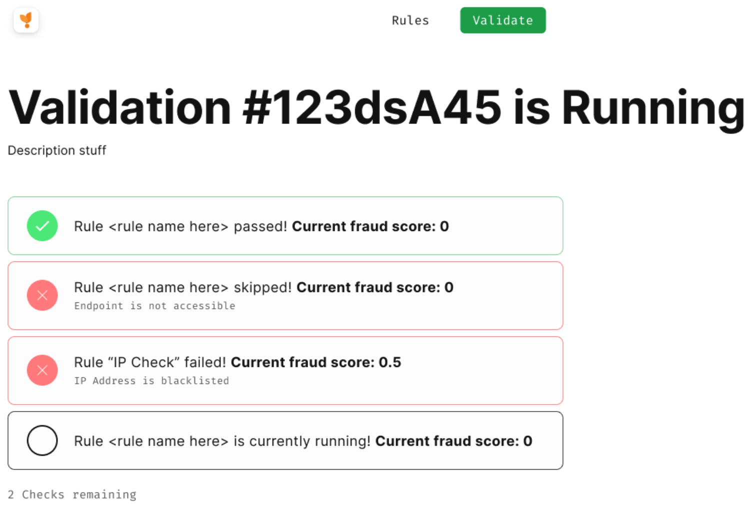

Figure 4.12.: *Mock up of the validation progress page*

in a list. The page gives the user information regarding the evaluation result of each validation rule (success, failure, in progress or not started), the current fraud score and the names of validation rules that are skipped. If present, the messages of an event should also be displayed.

5. Implementation

As the system design is made and the functionalities of each component is defined, the next step is the actual implementation of the system. This chapter describes the detailed information on how the structure and functionalities listed in main chapter 4 are implemented.

5.1. Technologies and Architecture

Before implementing a specific functionality of the system, each component of the system needs to be setup and configured, so that an iterative development process can be done correctly. For certain components of the system, a live environment is configured, so that the system is accessible via a public URL, without having to set it up locally.

5.1.1. User Interface

As mentioned in subchapter 4.3.1, the UI is a web application, built using Vue3 and TypeScript. The subsection describes some technical details of the UI project.

Build Tool

Nowadays, most front end projects use some kind of build tool to help build, test, and develop web applications[1]. *Vite*[2] is used as the build tool for the UI project.

Component Library

A component library is used in this project to speed up the development of the UI. The component library used in this project is *NaiveUI*[3]. NaiveUI is chosen because it contains several components that are suitable for the project (for example: `Timeline`, `Menu`) and it also supports TypeScript out of the box.

Code Style

To enforce a consistent code style and comply to the best practices of a TypeScript project, a code formatter (*Prettier*) and linter (*ESLint*) are used in this project.

[1] For detailed information on the role of a build tool in front end development, please take a look at [11].
[2] *Vite* is an open source front end build tool. GitHub repository: `https://github.com/vitejs/vite`.
[3] *NaiveUI* is a Vue3 component library. GitHub repository: `https://github.com/TuSimple/naive-ui`.

Docker

The UI can also run as a Docker container. Below is the `Dockerfile` implementation for the UI project.

```
FROM node:16-alpine

RUN npm install -g http-server
WORKDIR /app

COPY package*.json ./

RUN npm install
COPY . .

RUN npm run build

EXPOSE 3000
CMD [ "http-server", "dist", "-p 3000"]
```

Listing 5.1: *Dockerfile for UI (Docker)*

CI/CD

A CI/CD process is configured within the project, to run certain actions on specific events that happen in the codebase. The tool used as a CI/CD platform is *GitHub actions*[4]. The configured CI/CD actions in this project are:

- Run test and build, when there's a new pull request (PR)[5] to `main` branch
- Run release and increase the version of the app, when there's a new commit to `main` branch

Deployment

A live environment is available for the UI, made possible by using *Netlify*[6]. Every change made to the `main` branch will be built and deployed on the Netlify platform automatically.

5.1.2. FDS

As described in subchapter 4.3.2, the FDS is a server side application, built with Node.JS and TypeScript. To build a REST API with ease, the Express.JS framework will also be used in this project. This subsection describes the technical details revolving the FDS project.

[4] *GitHub actions* is a CI/CD platform, created by GitHub and can be used in all repositories hosted on GitHub. Homepage: `https://github.com/features/actions`.

[5] *Pull request (PR)* is a request from a developer to merge certain changes on a dedicated branch to the main branch of a repository.

[6] *Netlify* is a hosting platform with a git-based workflow. Homepage: `https://www.netlify.com/`.

Database Connection with ORM

To set up a database connection with Prisma (the ORM library of choice for FDS), the Prisma package should be installed and configured in the current project. The configuration file for Prisma is the `prisma.schema` file. The `prisma.schema` file is used to define the database schema, the type of the database system as well as its URL.

```
datasource db {
  provider = "mongodb"
  url      = env("DB_URL")
}
```

Listing 5.2: *Configuring database type and URL (Prisma)*

In this particular example, the database system for the current project is set to MongoDB, and the database URL will be provided by the `DB_URL` environment variable.

As the configuration is done and the database schema is created, the Prisma client API has to be updated to include the interfaces generated from the schema by running `npx prisma generate`.

Tsoa and Swagger

Even though Express.JS provides a declarative and easy way to build a server side application, it's built for a JavaScript application. Although TypeScript can be used with Express.JS, it doesn't use all the potential functionalities that TypeScript provides. *Tsoa*[7] is a framework with integrated openAPI compiler to build server side applications that can leverage TypeScript to its potential. Tsoa helps an express application to have the following functionalities out of the box:

- Generate Swagger[8] specification based on HTTP controller code
- Generate Swagger schema based on TypeScript interfaces
- Generate Swagger schema descriptions based on *jsDoc*[9] comments on the source code
- Validates an HTTP request body based on TypeScript interfaces

Tsoa provides an alternative syntax to build an Express.JS application in a more object-oriented way. Tsoa works by compiling the code, using the TypeScript and openAPI compiler into a regular Express.JS application.

Code Style

Identical to the UI project, a code formatter (*Prettier*) and linter (*ESLint*) are also used in this project. The configuration for the code formatter and linter is slightly different in comparison to the UI project, as certain code style rules doesn't apply to a server side application[10].

[7] *Tsoa* GitHub repository: `https://github.com/lukeautry/tsoa`.
[8] *Swagger* is a tool to document server side APIs. Homepage: `https://swagger.io/`.
[9] *jsDoc* is a tool to generate API documentation, similar to *JavaDoc*. Homepage: `https://jsdoc.app/`.
[10] In the UI project, there are certain code style rules regarding HTML elements.

Docker

The application will be built and run as a Docker container. A `Dockerfile` is provided, containing a list of commands needed to be run to assemble the particular image.

```
FROM node:16-alpine

ARG ARG_1 # Arguments passed by --build-args flag
ENV ARG_1=$ARG_1 # Environment variable of the image

WORKDIR /app
COPY ["package.json", "package-lock.json*", "./"]

RUN npm ci # Install packages
COPY . .
# Additional steps to setup the application

RUN npm run build
ENV NODE_ENV=production

CMD [ "node", "./dist/src/main.js" ]
```

Listing 5.3: *Dockerfile for FDS (Docker)*

CI/CD

A CI/CD process is also configured for this project using GitHub actions. The actions configured in this project are:

- Run test and build, when there's a new PR to the `main` branch
- Run release, bump version of the application and deploy it to the live environment, when there's a new commit to `main` branch

Deployment

A live environment is available for the FDS application. The FDS is deployed and run as a Docker container in *Heroku*[11]. The deployment process will be executed via the CI/CD action on every commit to the `main` branch.

5.1.3. RabbitMQ

A RabbitMQ instance is required as one of the integral parts of the system. Fortunately, RabbitMQ provided an official Docker image for it and running a RabbitMQ instance as a Docker container is as simple as running the following command in a shell terminal:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
    rabbitmq:3.10-management
```

Listing 5.4: *Running a RabbitMQ instance with Docker (Shell)*

[11] *Heroku* is a platform as a service (PaaS) that provides a platform for developer to build and run application in the cloud. Homepage: `https://heroku.com`.

The command listed above will locally run the RabbitMQ instance on port `5672` and the RabbitMQ management UI on port `15672`. On the live environment, a service called *CloudAMQP*[12] is used to help in setting up a RabbitMQ instance in the cloud, so that it can be accessed by other components via its public URI easily.

5.1.4. Redis

Redis is used in the system as a caching memory and a temporary data store. Redis also provided an official Docker image. To run a Redis instance as a Docker container, the following command should be run in a shell terminal:

```
docker run -d --name redis-stack-server -p 6379:6379 redis/redis-stack
    -server:latest
```

Listing 5.5: *Running a Redis instance with Docker (Shell)*

The command listed above will locally run the Redis instance on port `6379`. Redis is not available on the live environment. Therefore, no caching is available and an in memory data store (a basic JavaScript class) is used to replace Redis as the temporary data store.

5.1.5. MongoDB

MongoDB is the database of choice for the system. MongoDB has an official Docker image that can be used to run the database instance in a Docker container. Prisma requires that the MongoDB instance connected to the ORM to be configured as a replica set, so that partial writes on nested queries can be prevented [16]. To configure the replication of a MongoDB instance, additional steps are needed and an example Dockerfile from the Prisma official GitHub repository is used [12][13].

On the live environment, a service called *MongoDB Atlas*[14] is used to help host a MongoDB instance in the cloud, making it more accessible by other components of the system.

5.1.6. Docker Compose

Docker Compose is used to as a tool to start and set up the components of the system as Docker containers using a single `docker-compose up -d` command. To use Docker Compose, a configuration file is needed in a directory that contains the FDS and the UI project[15].

The environment variables of the FDS, such as the database URI and the RabbitMQ URI can be configured from the Docker Compose configuration file, making it easier to

[12] *CloudAMQP* is a service provider (RabbitMQ as a Service) that offers RabbitMQ clusters setup and management.

[13] The source code taken from the official GitHub repository is listed on Listing A.2.

[14] *MongoDB Atlas* is a service that provides a cloud-hosted MongoDB instances. Homepage: `https://www.mongodb.com/atlas`.

[15] Docker Compose configuration file is listed on Listing A.3.

configure the current context of the system without having to introduce any change to the existing code base.

```yaml
environment:
  - ENABLE_CACHE: true
  - DATA_STORE: "redis"
  - DATABASE_URL: "mongodb://root:prisma@mongo:27018/tests?authSource=admin/"
  - REDIS: "redis://redis:6379/"
  - RABBITMQ_URL: "amqp://rabbit:5672"
  - RABBITMQ_MANAGEMENT_UI: "http://rabbit:15672"
```

Listing 5.6: *Configuring environment variables of FDS (YAML)*

Running all the components of the system using a single command as Docker containers is useful, because it provides a solution on the added complexity of installing the needed dependencies and running each component individually.

Unfortunately, even though each component of the system is started by using the `docker-compose up -d` command, doesn't mean that every component is ready to be used directly. The RabbitMQ component needs a while to boot up, and therefore the FDS cannot yet be started until RabbitMQ is ready to accept any ongoing connection. Therefore, an additional function is implemented on the FDS to make sure that RabbitMQ is ready before attempting to connect, by waiting for the RabbitMQ management UI[16] to respond in a given time period.

5.2. Object Models

In this chapter, the specific implementation of the object models described in subchapter 4.4.2 will be discussed.

5.2.1. Validation Rule

The subsection describes the implementation of the `ValidationRule` object model as a TypeScript interface.

Retry Strategy

The `retryStrategy` attribute of the validation rule model is very specific to the implementation of the FDS. The FDS is using a library called *Got*[17] to make HTTP requests to the external endpoints. Got provides the functionality to retry a failed HTTP request out of the box[18].

The `retryStrategy` attribute of a validation rule is a subset of the retry options provided by Got's retry API, and will be passed to the Got instance when the HTTP request is made for the corresponding rule evaluation.

[16]The RabbitMQ management UI is available via a RabbitMQ plugin, also provided by the official Docker image.

[17]*Got* is a Node.JS library to make HTTP requests. GitHub repository: `https://github.com/sindresorhus/got`.

[18]For more information on Got's retry options, please refer to `https://github.com/sindresorhus/got/blob/main/documentation/7-retry.md`.

TypeScript interface

As the implementation of the model defined in Figure 4.8, a TypeScript interface is created to provide a clear structure of a validation rule.

```typescript
export interface ValidationRule {
  retryStrategy?: RetryStrategy | null
  requestUrlParameter?: GenericObject
  requestHeader?: GenericObject
  skip: boolean
  requestBody?: GenericObject
  condition: Condition | BooleanCondition
  method: "GET" | "PUT" | "POST"
  failScore: number
  endpoint: string
  priority: number
  name: string
}

type GenericObject = { [key: string]: any } // Dictionary
type Condition = {
  path: string
  operator: OperatorType
  type: ConditionType
  value: any
  failMessage: string
}

type BooleanCondition = {
  all: Condition[]
} | {
  any: Condition[]
}

type RetryStrategy = {
  limit: number
  statusCodes: number[]
}

```

Listing 5.7: *TypeScript interface of a validation rule (TypeScript)*

5.2.2. Validation Result

A TypeScript interface is created as the implementation of the validation result structure listed on Figure 4.9. The FDS is not responsible in storing validation results in a database. Therefore, a Prisma schema won't be created for validation results.

```
export interface Validation<T> {
  validationId: string
  fraudScore: number
  totalChecks: number
  runnedChecks: number
  skippedChecks: string[]
  additionalInfo: ValidationAdditionalInfo<T>
  events: ValidationEvent[]
}

export type ValidationEventStatus = "NOT_STARTED" | "FAILED" | "PASSED" | "RUNNING"
export type ValidationEvent = {
  name: string
  status: ValidationEventStatus
  dateStarted: string | null
  dateEnded: string | null
  messages?: string[]
}

export type ValidationAdditionalInfo<T> = {
  startDate: string
  endDate?: string
  customerInformation?: T
}
```

Listing 5.8: *TypeScript interface of a validation result (TypeScript)*

5.3. Features

This chapter describes the implementation of the features elicited on the previous chapters in details, specifically within the FDS component. All the HTTP routes of the FDS will be prefixed with `/api/v1`.

5.3.1. Validation Rules Management

The management of validation rule entries in the database are handled by the ORM library of choice (Prisma). This includes creating, reading, updating and deleting validation rule entries in the database. A Prisma schema[19] is created to specify the shape of the data saved in the database. The schema must be created beforehand to generate the required types to be used by the client. After the schema is created, the Prisma Client API should be updated by running `npx prisma generate`. The ORM simplifies the access to the database entries by providing type-safe interfaces and a layer of abstraction on top of the database system.

[19]The *Prisma* database schema is listed on Listing A.1

```typescript
import { PrismaClient } from '@prisma/client'

const main = async () => {
  const prisma = new PrismaClient() // Create new prisma instance
  await prisma.$connect() // Establish connection to database
}

```

Listing 5.9: *Establishing database connection with Prisma (TypeScript)*

```typescript
await prisma.validationRule.findMany()
await prisma.validationRule.delete({
  where: {
    name: validationRule.name
  }
})

```

Listing 5.10: *Example usage of Prisma (TypeScript)*

5.3.2. Customer Validation on a Registration Event

An HTTP endpoint is implemented to provide the possibility to schedule a validation process as soon as a new customer is registered. The endpoint should accept the customer information on the request body and return validation ID and additional information of the validation process as a response.

```typescript
@Example<ValidationSchedule>(validationSchedule)
@SuccessResponse(201, "Validation started")
@Response<ValidationErrorJSON>(422, "Validation Failed")
@Response<WentWrong>(400, "Bad Request")
@Post()
public async validateCustomer(
    @Body() requestBody: Customer,
  ): Promise<ValidationSchedule | WentWrong> {
    const { error, data } =
      await ValidationService.scheduleRulesetValidation(
        requestBody,
      )
    if (error) {
      this.setStatus(400)
      return {
        message: error.message,
        details: error.details || "",
      }
    }

    return data
}

```

Listing 5.11: *HTTP controller to schedule a validation process (TypeScript)*

The HTTP controller is intentionally kept as simple as possible. The logic behind the process to schedule a validation is done by the `ValidationService` and `ValidationEngine` (discussed in subchapter 5.3.3). The `ValidationService` is responsible in this particular case to get the lists of existing validation rules and runtime secrets, then creating a new instance of `ValidationEngine` as well as scheduling a new validation process.

5.3.3. Validation Process

The subsection gives a detailed explanation on how a validation process is structured, and the important components of the engine that runs a validation process.

Validation Process Flow

A validation process is started by iterating through a list of validation rules, making an HTTP request to the external endpoint listed on each rule and evaluating its response in comparison to the conditions attached on the rule. If the HTTP response from the external matches the conditions of the rule, the rule evaluation will be considered as a passed evaluation, otherwise it is a failed evaluation. The result of each rule evaluation determines the value of the resulting fraud score. The resulting fraud score will be calculated as follows:

- Initialize an empty list of fraud scores. The list will be filled later with float numbers ranging between 0 and 1
- Go through the list of validation rules and run evaluation
- If the evaluation passed, append `0` to the list of fraud scores
- Otherwise, append the validation rule `failScore` attribute's value to the list of fraud scores
- At the end of the iteration, the list size should equal to the amount of available[20] validation rules
- The resulting fraud score is the sum of the scores in the list, divided by the number of available validation rules

The flow listed above will be executed by creating a `ValidationEngine` instance and calling a public `secheduleRulesetValidation` method[21]. Before running a validation process, the list of validation rules as well as runtime secrets[22] should be provided to the engine. The `ValidationEngine` class uses method chaining[23] as well as the *Builder* design pattern discussed in [3, pp. 97-106] for its construction, to make the public API of a validation engine as simple as possible.

[20]*Not skipped.*
[21]Please take a look into Figure A.2 to see a flow diagram for the validation process.
[22]Runtime secrets are used to store confidential information such as API keys.
[23]*Method chaining* is a way to provide the possibility of invoking multiple method calls of an object without having to store an intermediary result in an additional variable.

```typescript
export class ValidationEngine<T> {
  private secrets: GenericObject = {}
  private ruleset: ValidationRule[] = []

  setSecrets(secrets: GenericObject) {
    this.secrets = secrets
    return this
  }

  setRuleset(ruleset: ValidationRule[]) {
    this.ruleset = [...ruleset.sort((a, b) => b.priority - a.priority)]
    return this
  }
}

```

Listing 5.12: *ValidationEngine class builder pattern using method chaining (TypeScript)*

Accessing Data by Evaluating JSONPath Expressions

To be able to access essential information stored in the current runtime scope a validation process, a JSONPath expression can be used in certain attributes of a validation rule. The provided runtime information of a validation process includes the customer information, runtime secrets and during a *condition* evaluation, the HTTP response from an external endpoint.

The ability to access certain information from the runtime scope is needed before making an HTTP request and when evaluating a condition. To evaluate a JSONPath expression, the `jsonpath` library is used.

```typescript
import jp from "jsonpath"

const accessDataFromPath = (runtimeData: any, expression: any) => {
  if (typeof expression !== "string") {
    return expression // A valid JSONPath expression is a string
  }

  try {
    const [dataFromPath] = jp.query(runtimeData, expression)
    if (!dataFromPath) {
      // Expression is valid, but the path doesn't point to a specific value
      return expression
    }

    return dataFromPath
  } catch {
    return expression // The expression is not a valid JSONPath expression
  }
}

```

Listing 5.13: *Accessing runtime information using JSONPath expression (TypeScript)*

Making an HTTP Request to External Endpoints

A dedicated class (`Agent`) is created to make an HTTP request to the external endpoint. The class will provide a layer of abstraction on top of the Got library that is being used to actually make the HTTP requests.

The class will also help in setting the request body, request header as well as to change the variables on the `endpoint` attribute with its corresponding values. The class follows the *Singleton* design pattern described in [3, pp. 127-134], as there is only one instance needed for the whole application. The class is also implemented using the dependency injection in mind, for an easier access to the underlying library during the testing phase. The dependency injection is implemented by providing a `context` object to the `Agent` class beforehand. During runtime, the `context` object contains the actual Got instance, while during testing, the Got instance is mocked.

```typescript
export class Agent {
  private static context: Context // Dependency injection

  private static get client() {
    return Agent.context.client // `client` object is a Got instance
  }

  static setClient(context: Context) {
    this.context = context
  }
}

```

Listing 5.14: *Usage of the singleton pattern and dependency injection in Agent class (TypeScript)*

Operators

`Operators` are special classes that define the operation of a certain condition during a validation process. Each `Operator` is grouped by its type and has two main properties; `identifier`, and `operateFunction`

The `identifier` property of an operator refers to the operator's name, unique in its type group. The `identifier` attribute of an operator will be passed into the `operator` attribute of a condition to describe the specific operator to be used in evaluating the particular condition. The `operateFunction` attribute of an operator is a function that accepts two arguments, and returns a boolean value that indicates whether the operation is successful. An additional validation process is also implemented using the `validateFunction` property to make sure that the value being passed into the `operateFunction` of an operator is valid. The `identifier` and `operateFunction` attributes of an operator are passed into the object constructor during its initialization, while the `validateFunction` attribute of an operator is defined by each subclass of the operator, grouped by its type.

```
export class NumberOperator extends Operator<number, NumberOperatorIds> {
  const validateFunction = (value) =>
    typeof value === "number" &&
    !isNaN(parseFloat(`${value}`))
}

export const numberOperators: Record<NumberOperatorIds, NumberOperator> = {
  eq: new NumberOperator("eq", (a, b) => b === a), // equals
  gt: new NumberOperator("gt", (a, b) => b > a), // greater than
  gte: new NumberOperator("gte", (a, b) => b >= a), // greater than equals
  lt: new NumberOperator("lt", (a, b) => b < a), // lesser than
  lte: new NumberOperator("lte", (a, b) => b <= a), // lesser than equals
}
```

Listing 5.15: *NumberOperator example (TypeScript)*

The *Flyweight* design pattern mentioned in [3, pp. 195-206] is used here, by instantiating all the available operators beforehand, and using the instantiated object during a condition evaluation. For an even easier access to the operators, a *flyweight factory* is also created. The `OperatorFactory` will return the appropriate operator to be used based on the type and identifier passed. If the combination of type and identifier of an operator doesn't point into a specific operator, a `NullishOperator` will be returned, which always return `false` as its operation result.

Evaluating a Rule

To evaluate a certain validation rule, the `Evaluator` class is created. The `Evaluator` class is responsible in running an evaluation on a certain condition by working together with the `Operator` class. The specific operator to evaluate a condition is accessed via the `OperatorFactory`. The `Evaluator` class encapsulates the internal logic of evaluating a condition by accessing the required data described by a JSONPath expression from the runtime information and running the operation defined by the particular condition's `type` and `operator` attributes.

```
// Evaluate JSONPath expressions.
const dataFromPath = this.accessDataFromPath(runtimeData, validationRule.path)
const valueFromPath = this.accessDataFromPath(runtimeData, validationRule.value)

const operator = OperatorFactory.getOperator(
  validationRule.type,
  validationRule.operator,
)
const isEvaluationPassed = operator.operate(valueFromPath, dataFromPath)
```

Listing 5.16: *The usage of OperatorFactory class in the Evaluator class (TypeScript)*

A condition can either be a single condition, or multiple conditions, wrapped inside an ***ANY*** or ***ALL*** attribute. The evaluation of a single condition is different from the evaluation of multiple conditions. To facilitate the different logic of evaluating conditions, two subclasses of the `Evaluator` class is created. `ConditionEvaluator` evaluates a single condition, while the `BooleanConditionEvaluator` is responsible in evaluating multiple conditions and handling the logic behind evaluating the ***ANY*** and ***ALL*** modifier. To simplify the instantiation of the suitable `Evaluator` subclass, the `EvaluatorFactory` class is created, following the *Factory Method* design pattern described in [3, pp. 107-116].

```
const response = await Agent.fireRequest(rule, {
  customer: customerData,
  secrets: this.secrets
})

const evaluator = EvaluatorFactory.getEvaluator(condition)
const evaluationResult = evaluator.evaluate({
  response: response.data,
  customer: customerData,
  secrets: this.secrets
})

```

Listing 5.17: *EvaluatorFactory usage in ValidationEngine class (TypeScript)*

As the evaluation result, an `Evaluator` instance will return an object with the `pass` attribute to determine whether the evaluation passed and an additional `messages` attribute, which contains essential information regarding the evaluation (including the value of `failMessage` attribute of a condition, if the evaluation failed).

5.3.4. Notification on Suspicious Cases

In [15], Subramoni et al. describes an AMQP *exchange* as a routing mediator that copies and sends the message published by a message publisher to zero or more message queues. The FDS acts as the message publisher of the system and publishes a message to a pre-defined exchange on every completion of a validation process.

An interface to access and publish a message to the exchange is implemented using the *Singleton* [3, pp. 127-134] design pattern, to only have a single connection to the RabbitMQ, since an AMQP connection are designed to be long-lived and opening a new connection to a RabbitMQ instance is an expensive operation. The type of the exchange used in the system is the *fanout* exchange. When a message is published to a fanout exchange, the message will be routed to all the queues bound to the exchange, which is ideal for the use case of the current notification system.

```
import { Channel, connect } from "amqplib"
export class Notification {
  private static channel: Channel | null = null

  async init(url: string) {
    try {
      const connection = await connect(url)
      const channel = await connection.createChannel() // Create a new channel
      // Assert whether the exchange exists, create new if it doesn't exist
      await channel.assertExchange("FDS", "fanout", {
        durable: true
```

```
      })

      Notification.channel = channel
    } catch (err) {
      console.error(err)
    }
  }
}
```

Listing 5.18: *Openning a connection to RabbitMQ instance (TypeScript)*

The `Notification` class also provides the `publish` static method, which can be used to publish a new message to the exchange if the connection is opened successfully. The `ValidationEngine` class calls the `publish` method every time a validation process is completed, publishing the validation result of the particular validation process to the exchange.

```
Notification.publish(JSON.stringify(this.validationResult))
```

Listing 5.19: *Publishing a validation result to the RabbitMQ exchange (TypeScript)*

There can be many consumers consuming the messages published to the exchange, and the consumers can run certain actions based on their internal logic. The message consumer can, for example email the concerned parties if the fraud score exceeds a certain threshold or even automatically block a customer if a specific rule evaluation failed. To consume a message published to the exchange, a connection to the RabbitMQ instance should be opened, and a dedicated message queue (ideally created only for internal usage of the consumer itself) is required.

```
import { connect } from "amqplib"
export const start = async (url: string) => {
  try {
    const connection = await connect(url)
    const channel = await connection.createChannel()
    await channel.assertExchange("FDS", "fanout", { durable: true })
    // Create an exclusive queue (created only for internal usage)
    const { queue } = await channel.assertQueue("", { exclusive: true })

    // Bind the exclusive queue to the exchange
    channel.bindQueue(queue, "FDS", "")

    await channel.consume(queue, async (message: string) => {
      // Do actions
    }, {
      noAck: true
    })
  } catch (err) {
    console.error(err)
  }
}
```

Listing 5.20: *Consuming a message published to the RabbitMQ exchange (TypeScript)*

5.3.5. Database Connection

To manage the validation rules and runtime secrets, a database connection to modify the database entries via the ORM (Prisma) is required. The *Singleton* [3, pp. 127-134] design pattern is also used here to make sure, that only a single connection to the database is created. Dependency injection is also used here to be able to provide a mocked Prisma instance during testing.

Caching can also be enabled in the `Database` class, by providing a `DataStore` instance to the static `setCache` method. By enabling caching, the values retrieved and updated can be cached to provide a faster response time[24].

5.3.6. Validation Real Time Progress

The *Observer* [3, pp. 293-303] design pattern will be implemented to provide the functionality of a real time progress update of a running validation process. An `EventEmitter` is used here to subscribe to and publish certain events during a validation process. The FDS publishes a `validation_event:update` event whenever a rule evaluation is done, containing the validation ID attached to the event name and sending the current validation result on the payload. When the particular validation process is done, the FDS also publishes a `validation_done` event, containing the validation ID on event name, as an indicator that the validation process is completed and no further events with the attached validation ID will be sent.

```
// Published when a rule evaluation is completed
EventBus.emit(
  `validation_event:update--${validationId}`,
  this.validationResult,
)
// Published when a validation process is completed
EventBus.emit(`validation_done--${this.validation.validationId}`)

```

Listing 5.21: *Publishing events on certain validation events using the EventEmitter (TypeScript)*

To continually provide a client with the latest progress of a validation event without having to set up a dedicated instance of a WebSocket server, the Server-Sent Events (SSE)[13] technology is used. The SSE enables the possibility to send new data to the client multiple times by only opening a single HTTP connection.

The difference between SSE and the WebSocket protocol, is that SSE only enables a one-way communication from server to client, meaning the client cannot continually send any new messages to the server, while the WebSocket protocol provides a two-way communication. SSE is enough for the use case of the project, as the client doesn't need to send new data to the server to display real time progress of a validation process.

[24]Runtime secrets are not cached.

The SSE is implemented by setting the `Content-Type` header of the HTTP response to `text/event-stream` and setting the `Connection` header to `keep-alive`. The `Connection: keep-alive` header is useful to keep the connection between client and server open, while the `Content-Type: text/event-stream` header specifies the type of content sent to the client. A message written to the event stream is prefixed with a `"data: "` prefix.

After the header of the response is set, the FDS will push a new message to the stream whenever there's a new event emitted on the `EventEmitter` with the corresponding validation ID.

```
static async subscribeToValidationProgress (
  validationId: string,
  responseObject: Response,
) {
  const updateEvent = `validation_event:update--${validationId}`
  const closeEvent = `validation_done--${validationId}`

  EventBus.once(closeEvent, () => {
    closeConnection()
  })

  EventBus.on(
    updateEvent,
    (validationResult: Validation) => {
      writeToStream(validationResult)
    },
  )

  const writeToStream = (data: any) =>
    responseObject.write(
      `data: ${JSON.stringify(data)}\n\n`,
    )
}

```

Listing 5.22: *Writing to SSE stream when certain events are published (TypeScript)*

5.4. User Interface

This chapter discusses the implementation of the structure and functionalities of the UI described in subchapter 4.4.3.

5.4.1. Navigation

The user interface contains functionalities for several use cases, merged into a single web application. In a web application, routing plays a key role in determining what should be displayed based on the URL address entered on the browser. Specifically in this case, routing can help in categorizing the current context of the application that consists of several views for different use cases. The following routes are implemented in the UI:

- Home page (path: `/`)
- Create a new rule (path: `/rules/new`)
- Edit a rule (path: `/rules/:ruleName`)[25]
- List of rules (path: `/rules`)
- Create a new validation (path: `/validations/create`)
- See validation progress for specific validation ID (path: `/validations/:validationId`)
- List of validations (path: `/validations`)

To help the user in navigating through the pages of the UI, a header is created and displayed on every page of the application. The header includes three buttons (Home, Rules and Validations), that links the user to the corresponding page of the application.

Figure 5.1.: *Screenshot of the header to navigate the UI*

5.4.2. Rule Management Form

The rule management form is a reusable form, can be used to both create a new and edit an existing validation rule. The rule management form displays all the attributes of a `ValidationRule` model as a form field.

Conditions Section

The Conditions section is a component, used to add one more condition to a validation rule. The Conditions section renders the list of conditions provided in a card, containing form fields for each attribute of the particular condition. Each card represents a single form, which also contains a form validation mechanism on its fields.

The available values for the `operator` attribute of a condition depends on its `type` attribute. To prevent an invalid condition being sent to the FDS, the Operator field is a select field, and its options are defined by the current value inputted on the Type field. The intention of the restriction is to make sure that the `operator` chosen is always suitable with the corresponding `type` attribute of the condition.

If more than one condition is provided, a radio field is also rendered, so that the user can choose one of the provided modifier for the list of conditions (either ***ALL*** or ***ANY***).

An additional validation is implemented in the Conditions section. The validation not only make sure that all the required fields are filled, but also the value of the fields itself. For example, the validation will throw an error if the Type field is set to "Number", but the input value of the Value field is not a valid numerical value.

[25]*:ruleName* refers to a dynamic value of a rule's name. Please take a look into `https://router.vuejs.org/guide/essentials/dynamic-matching.html` for more information.

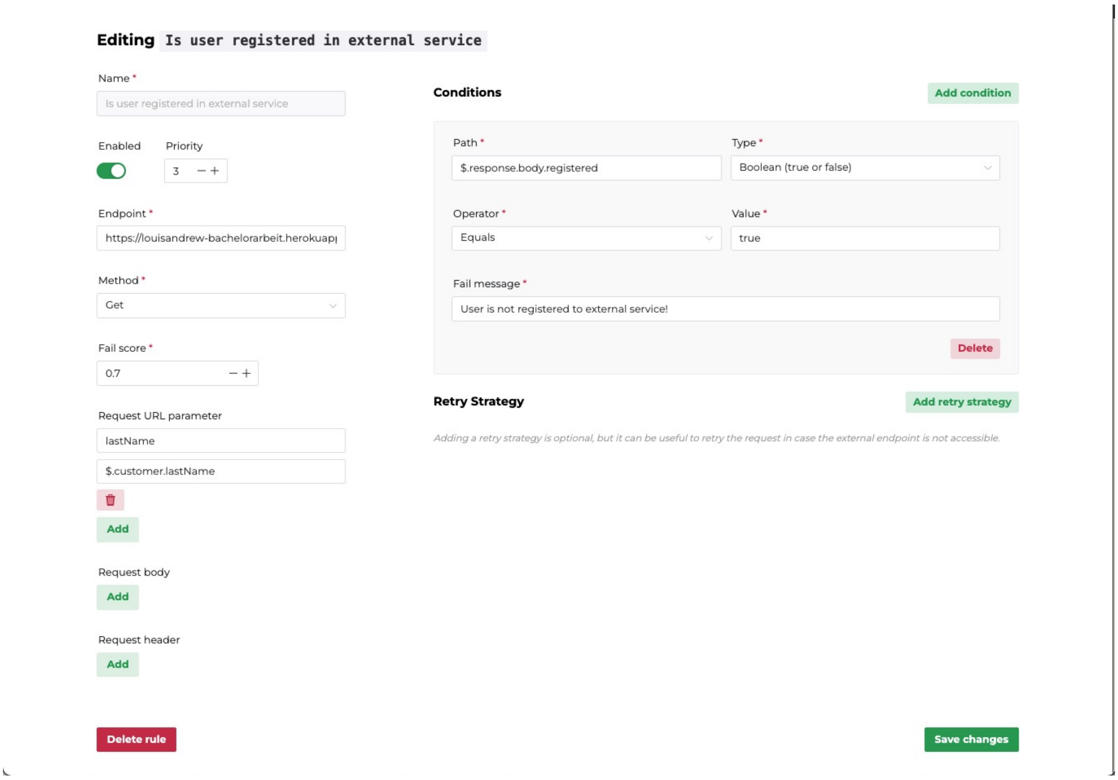

Figure 5.2.: *Screenshot of the rule management form*

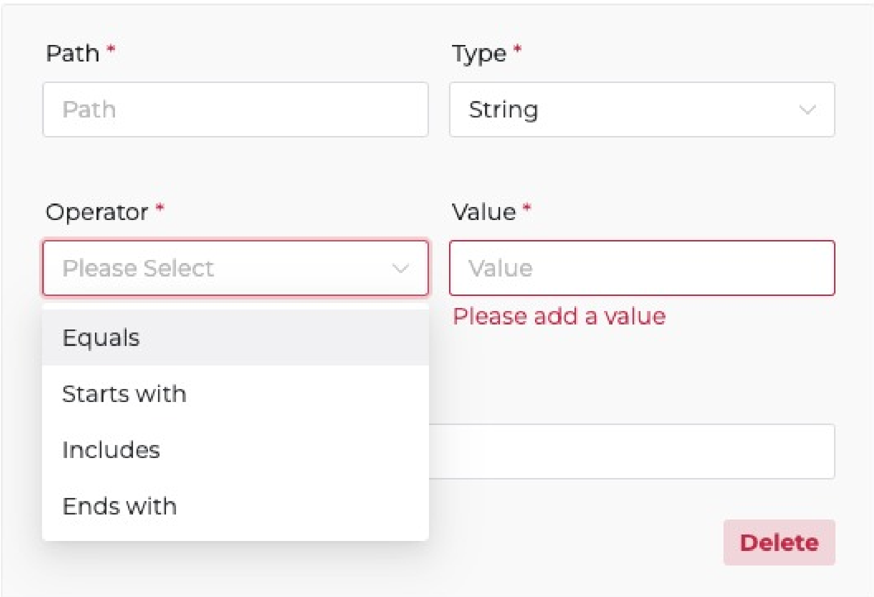

Figure 5.3.: *Screenshot of the "Operator" select field options, based on the current "Type" value*

Autocomplete Input

A JSONPath expression might be used as a value of some attributes within a `ValidationRule` to access the runtime information during a validation process, such as the HTTP response from the external endpoint, runtime secrets or customer information. Unfortunately, it might be difficult to memorize the expressions needed to access certain values, and it might also confuse the user.

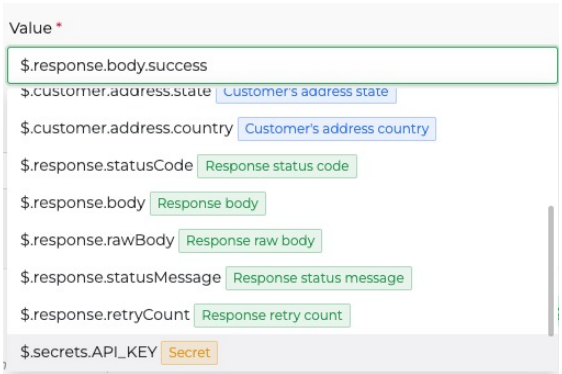

Figure 5.4.: *Screenshot of the autocomplete input usage*

To solve this problem, an autocomplete input field is provided in certain fields where a JSONPath expression is used. User can then display a list of possible JSONPath expressions by prefixing an input field with "$", and choosing one of the expressions listed. User can also extend the expression chosen. The autocomplete input is used in the following form fields:

- PATH field on CONDITIONS section
- VALUE field on CONDITIONS section
- Value field of a dynamic input[26]

5.4.3. Validation Form

The validation form is a simple form similar to a customer registration form, specifically to mimic a new customer registration and to run a validation process directly after a new registration. Form validation is also implemented in the validation form to make sure that the customer information sent to the FDS is complete. A list of sample customers with specific characteristic is provided to pre-fill the validation form with a sample data.

```
const applySampleCustomer = (sampleCustomer: Customer) => {
  Object.assign(formValues, {
    ...sampleCustomer,
  })
}

```

Listing 5.23: *Prefilling form values with a sample customer data (TypeScript)*

When the user filled the fields properly and clicked the VALIDATE CUSTOMER button, the UI sends an HTTP POST request to the FDS with the customer data as the payload to schedule a new validation process. As the FDS returns an ID of the validation

[26]Autocompletion on *$.response* is only available on the CONDITIONS section.

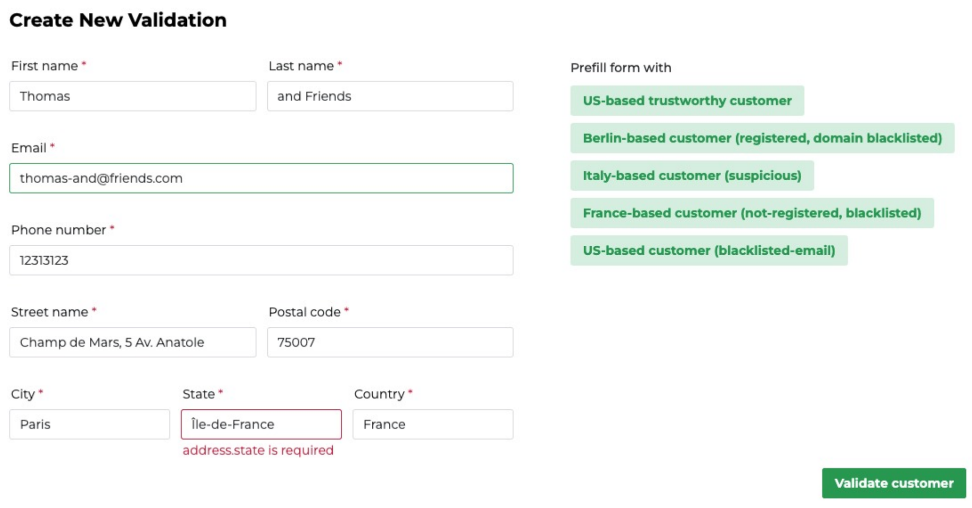

Figure 5.5.: *Screenshot of the validation form*

process, the UI will then redirect the user to the validation progress page, showing the progress of the particular validation process in real time.

```
const { data } = await createNewValidation(customer)
if (data.validationId) {
  router.push(`/validations/${data.validationId}`)
}

```

Listing 5.24: *Redirecting to the progress page when a validation process is scheduled successfully (TypeScript)*

5.4.4. Validation Progress

To receive the real time event stream sent by the FDS as described in subchapter 5.3.6, the `EventSource` API can be used on the client side. With the `EventSource` API, a persistent connection to the FDS will be opened, and the messages sent will be received by the client in real time. Because the FDS sends the messages as a string in an event stream, the UI has to parse the message content into a valid JSON object before processing it. It is also important to close the connection to the event stream when it's no longer needed. The logic of closing the open connection when user leaves the current page is implemented on the UI.

```
const source = new EventSource(url)

source.onmessage = messageEvent => {
  try {
    const data = JSON.parse(messageEvent.data)
    // Do data processing
  } catch {
    // Handle error, if the message data is not a valid JSON object
```

```
  }
}

```

Listing 5.25: *Using the EventSource API in the browser (TypeScript)*

As the content of the message is parsed and identified as a valid `ValidationResult` object, it is saved into the current state of the `Validation` component, which renders the events of a validation process into a timeline component to give the user a better graphical overview of the current process. Vue3 uses the *Proxy*[3, pp. 207-217] design pattern under the hood to establish a reactivity system on the UI, providing the possibility to update the timeline component every time a new validation result is published by the FDS.

The ID of the validation as well as the current fraud score are always displayed, to provide detailed information on the current validation process. Additionally, the customer data is also displayed as a JSON object, to give the user an insight on the customer data being sent to the FDS.

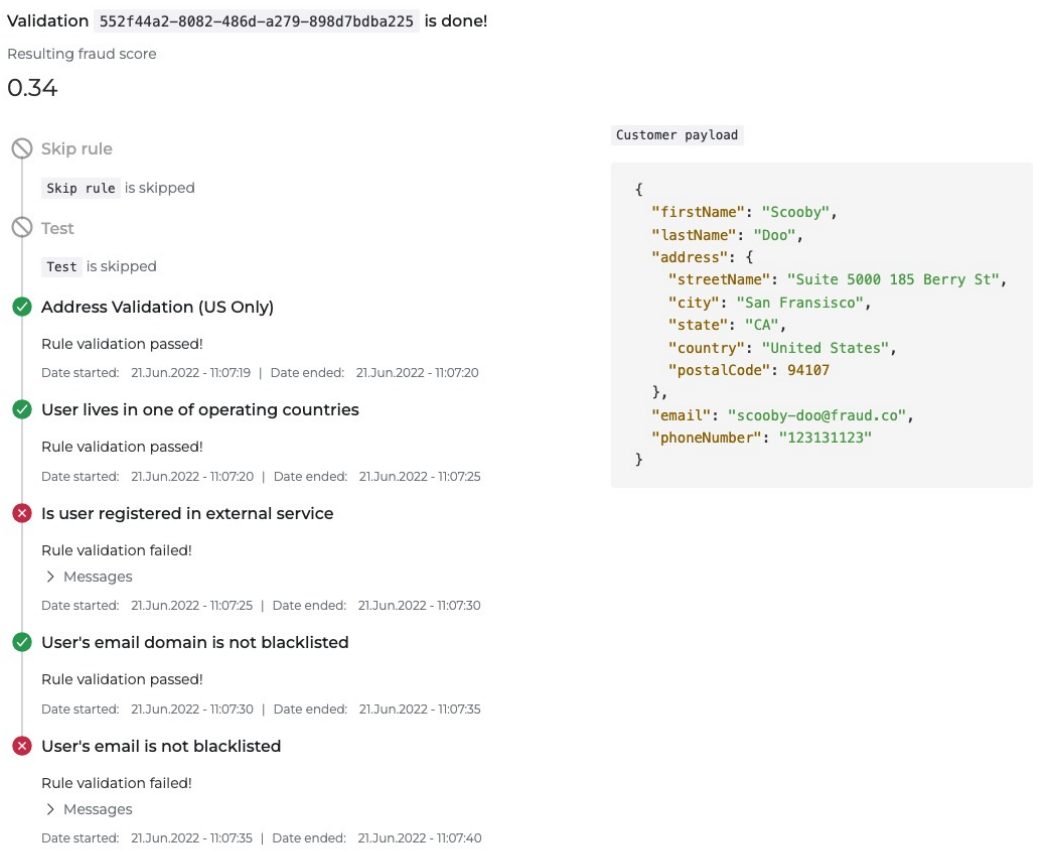

Figure 5.6.: *Screenshot of the validation progress in real time*

5.4.5. Runtime Secrets

A component to display a list of the available runtime secrets in a dialog is created. For security purposes, only the key of the secrets are displayed within the component. The component also provides a possibility to create a new runtime secret and to delete an existing secret. User can create a new secret by clicking on the CREATE A NEW SECRET button, and adding the essential information on the form fields displayed and delete an existing secret, a DELETE button is displayed next to each secret's key name. When a user creates a new runtime secret or deletes an existing secret, the autocomplete options are changed according to the latest list of keys available in the database.

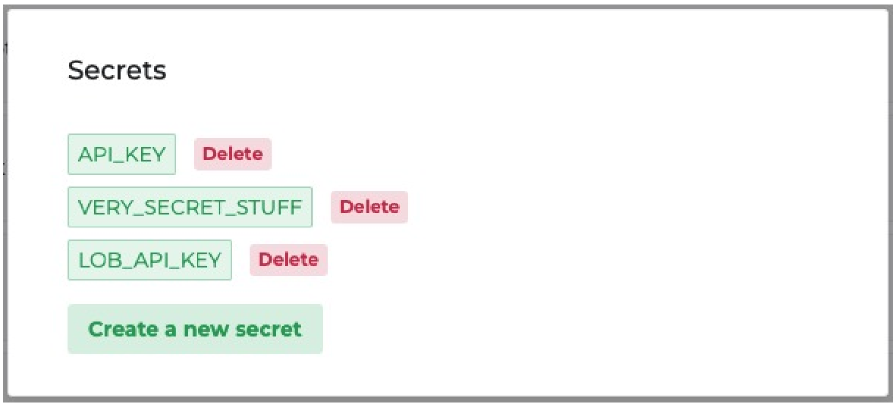

Figure 5.7.: *Screenshot of a component to list available runtime secrets*

6. Test

Testing is an integral part of software development. Writing tests gives the developer the confidence that the recent changes made will not break the remaining parts of the software. The system is built by writing the tests before implementing the particular features (also known as *TDD* method), in hope that the tests can help reduce the amount of bugs in the system while also providing a code-level working specification of the particular component. The technology used to run test suites for both the FDS and the UI is *vitest*[1]. This chapter discusses the tests written during the development process.

6.1. Unit Test

Unit tests play an important role during the development process, as it make sure that each component of the system works perfectly and fulfills its function in isolation.

6.1.1. FDS

The main component of the FDS, the `ValidationEngine` consists of many components. To make sure that even the slightest change to the codebase won't break the whole system, each component is tested in isolation to ensure its function. Dependency injection pattern is also used here as a way to mock underlying third party libraries used by the component.

```typescript
describe("Agent", () => {
  const mockContext: MockContext = createMockContext()
  Agent.setClient(mockContext)

  afterEach(() => mockReset(mockContext))

  it("fires an HTTP request to the correct endpoint", async () => {
    await Agent.fireRequest(sampleRule, {})

    expect(mockContext.client).toBeCalledWith(sampleRule.endpoint, expect.anything())
  })
})
```

Listing 6.1: *Dependency injection usage in a unit test within FDS project (TypeScript)*

[1] *Vitest* is a unit test framework, with out-of-box TypeScript support and Vite native compatibility. GitHub repository: `https://github.com/vitest-dev/vitest`.

```typescript
describe("Condition Evaluator", () => {
  const condition: Condition = {
    path: "$.statusCode",
    operator: "eq",
    type: "number",
    value: 200,
    failMessage: "Status code doesn't equal to 200",
  }
  const evaluator = new ConditionEvaluator(condition)

  it("returns the correct result for a valid data", () => {
    const { pass, messages } = evaluator.evaluate({
      statusCode: 200,
    })

    expect(pass).toBeTruthy()
    expect(messages).toEqual([])
  })
})

```

Listing 6.2: *Example unit test of the condition evaluator (TypeScript)*

6.1.2. UI

Testing a client-side web application might not be as straightforward in comparison to testing a server-side application. What's being tested in a client-side application is actually the behavior of each component and how it interacts to the user input.

Unit testing on the UI is done by isolating the smallest component and making sure it is behaving properly according to its specification. An additional library is needed to simulate the browser runtime-environment during testing. The library *Vue testing library*[2] is chosen as it provides several APIs to test the behavior of a UI component by resembling on what the user actually sees in the browser.

```typescript
describe("Key-Value input", () => {
  const renderComponent = (options: RenderOptions) => render(KeyValueInput, options)

  it("renders a new key-value input fields when `Add` is clicked", async () => {
    const { getByRole, queryAllByTestId } = renderComponent({})
    const addButton = getByRole("button", {
      name: "Add"
    })

    expect(queryAllByTestId("key-value-field").length).toBe(0)
    await fireEvent.click(addButton)

    expect(queryAllByTestId("key-value-field").length).toBe(1)
    await fireEvent.click(addButton)

    expect(queryAllByTestId("key-value-field").length).toBe(2)
```

[2] *Testing library* is a family of packages for several front-end frameworks built to help test UI components. Homepage: `https://testing-library.com/`

```
  })
})

```

Listing 6.3: *Example unit test of a UI component (TypeScript)*

6.2. Integration Test

Integration testing is used to make sure that individual components of the system can be integrated well and functions properly in unity.

6.2.1. FDS

The integration testing on the FDS is done by making a mocked HTTP request to the HTTP endpoints provided and evaluating its result. The library *supertest*[3] is used to mock the HTTP request during the integration tests. Before running an integration test, the whole application has to be set up in a testing environment, to make sure that all the components of the system are ready.

```
export const initTestingServer = async () => {
  const mockContext = createMockContext()
  const store = initStore("in-memory")
  const database = new Database(mockContext)

  await store.init()
  await database.init()

  return { app, mockContext, store }
}

```

Listing 6.4: *Setting up a testing server environment for integration test (TypeScript)*

The database connection is mocked during integration testing, to prevent unintentional mutation of the data stored in the database and to prevent any outgoing network requests during the automated testing process. The integration tests are written to make sure that all the components, from the HTTP controller to the underlying services that retrieve the data work correctly as a whole.

```
describe("Rules CRUD endpoint", () => {
  let agent: SuperTest<Test>
  let mockContext: MockContext

  beforeAll(async () => {
    const { app, mockContext: ctx } = await initTestingServer()
    agent = request(app)
    mockContext = ctx
  })
```

[3] *Supertest* is a library to test Node.JS HTTP servers. GitHub repository: `https://github.com/visionmedia/supertest`.

```
  it("GET /rules/:ruleName should return a single rule", async () => {
    mockContext.prisma.validationRule.findFirst.mockResolvedValueOnce(
      prismaValidationRule,
    )

    const response = await agent.get(
      "/api/v1/rules/" + prismaValidationRule.name,
    )
    expect(response.statusCode).toEqual(200)
    expect(response.body).toEqual({
      id: "", ...sampleRule
    })
  })
})
```

Listing 6.5: *Integration testing on the FDS (TypeScript)*

6.2.2. UI

Integration test on the UI is written to make sure that each UI components work collectively in a correct way. Integration tests for a client-side web application is particularly useful in making sure that the interaction between each component works well and behaves exactly as it should. The code snippet listed below represents an integration test that verifies the `ConditionList` component is working properly in combination with the `RuleForm` component.

```
describe("Rule form component", () => {
  const renderComponent = (
    options: RenderOptions = {
      props: {
        rule,
      },
    }
  ) => render(RuleForm, options)

  it("emits the correct event when the rule is updated", async () => {
    const { getByPlaceholderText, getByRole, emitted } = renderComponent()

    await fireEvent.update(getByPlaceholderText("Path"), "XX")

    await fireEvent.click(getByRole("button", { name: "Save changes" }))
    await waitFor(() => expect(emitted()["update"]).toBeTruthy())
    const newRule = (emitted()["update"][0] as ValidationRule[])[0]
    expect(newRule.condition.path).toEqual("XX")
  })
})
```

Listing 6.6: *Integration test on the UI (TypeScript)*

7. Demonstration and Evaluation

After the features of the system are implemented and tested, the demonstration as well as the evaluation of the system as the result of this research project can be done. This chapter presents the demonstration of the use cases elicited from the goal of the research project as well as its evaluation based the software quality standards described in subchapter 3.4.3.

7.1. Demonstration

In this chapter, the following procedure will be demonstrated as the result of this research project:

- Retrieve a list of existing validation rules
- Creating a runtime secret (API key) to be used in one of the validation rule
- Create a new validation rule
- Retrieve a list of validation processes
- Schedule a new validation process
- Email a specified email address if the fraud score of the validation exceeds `0.5`

For demonstration purposes, an AMQP consumer is implemented to email a specified email address if the fraud exceeds a certain number[1]. Furthermore, an external address verification API[2] will be used as the external service for this demonstration. Therefore, an API key for the corresponding API needs to be created and added to the runtime secret of the validation engine.

After running the application using the command `docker-compose up`, the UI is accessible on `http://localhost:3000` and the FDS is accessible on `http://localhost:8000`. The URL `http://localhost:3000/#/rules` can be visited to display the list of available validation rules in the database. A new runtime secret can be created from the rules list page by clicking the Secrets button and then Create a new secret button afterward.

The lob API uses the *Basic Auth* authentication scheme, which requires a client to send the `Authorization` header that contains the `"Basic "` prefix, followed by the base64 encoded value of the Lob API key. As the runtime secret is created, it is now accessible to the validation rule using the `$.secrets.LOB_API_KEY` JSONPath expression. The creation of a new validation rule can be started by clicking on the Add new rule button. A rule management form is displayed, and the user can enter the values to each attribute of the validation rule. For the demonstration, the following validation rule is created:

[1]The AMQP consumer for the demonstration purpose is available as an attachment to this thesis

[2]The external address validation used for the demonstration is Lob address verification. Homepage: `https://www.lob.com/address-verification`.

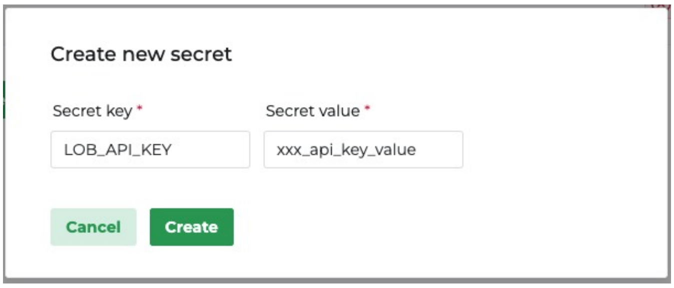

Figure 7.1.: *Screenshot of a runtime secret creation*

- Name: "Address Validation"
- Endpoint: `https://api.lob.com/v1/intl_verifications`
- HTTP method: `POST`
- Fail score: 0.5
- Request body:
  - `recipient: "FDS"`
  - `primary_line: $.customer.address.street`
  - `city: $.customer.address.city`
  - `state: $.customer.address.state`
  - `country: $.customer.address.country`
- Request header: `Authorization: Basic $.secrets.LOB_API_KEY`
- Conditions:
  - Evaluate whether the status code equals to 200
  - Evaluate whether the response body returns a `valid_address` attribute and whether it equals to `true`

By visiting the URL `http://localhost:3000/#/validations`, the list of ongoing and completed validation processes, saved in the data store of the FDS will be displayed. A new validation process can be created by clicking on the CREATE NEW VALIDATION button and filling in the validation form with a sample customer data, on which a validation process should be executed. As mentioned before, several sample customers with different attributes are created to provide an even easier testing process. For this demonstration, the user with the label `Berlin-based customer (invalid address)` will be chosen, as it will trigger a failed rule evaluation of the `Address Validation` validation rule.

After filling the validation form with a sample customer data, a validation process will be scheduled by clicking on the VALIDATE CUSTOMER button. As a validation process is scheduled, the user will be redirected to a validation process page, on which the progress of the validation process will be displayed in real-time.

Upon the completion of the validation process, validation result should return a `0.5` fraud score, as the sample customer's address is invalid. As mentioned earlier, a certain action should be done by the AMQP consumer, when there's a validation process that resulted in a fraud score of `0.5`. An email should be sent to the email address, specified when running the AMQP consumer.

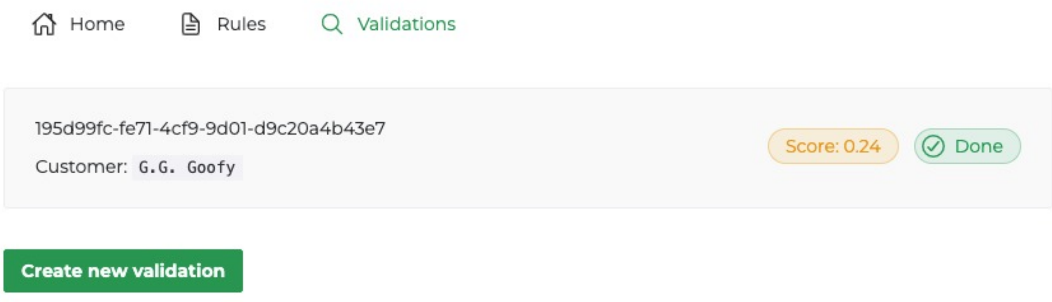

Figure 7.2.: *Screenshot of the validation list page*

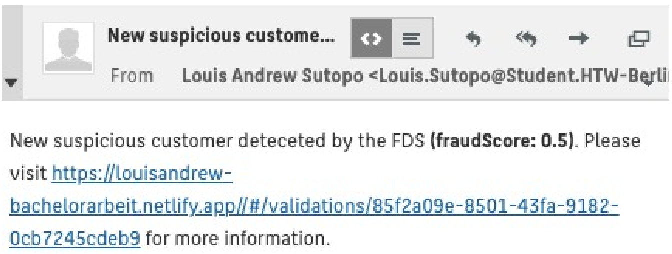

Figure 7.3.: *Screenshot of an email sent by AMQP consumer when a validation process completed with a fraud score exceeding 0.5*

7.2. Evaluation

The completion of the goals and criteria described in requirement analysis should be evaluated as a measure to determine whether the system and its functionalities have accomplished its main objective according to the goal of the research project. The implementation of each user stories described in subchapter 3.4.2 will be listed in the table below:

Table 7.1.: *Implementations of the primary use cases of the system*

User Story	Implementation
As a stakeholder, I want to verify users, so that the company can have more confidence that the existing user base is trustworthy	Providing an HTTP endpoint to trigger a validation process based on its request body and asynchronously returning a validation result that contains a numerical value to determine the probability of the customer being a fraud
As an employee, I want to be notified when a user seems suspicious, so that I can do necessary actions accordingly	Providing an AMQP message broker and publishing the message to an AMQP exchange whenever a validation process is completed. An additional AMQP message consumer can be implemented to send the necessary notification

As an employee, I want to manage my own rule to validate users, so that I can use my expertise to find suspicious customers as efficiently as possible without the communication overhead with other teams	Providing endpoints to create, read, delete and update validation rules. Additionally, endpoints to create and delete runtime secrets to store confidential information (such as API keys) are also implemented

Furthermore, a further evaluation on the criteria listed in subchapter 3.4.3 should also be done, to determine the quality of the system created as the result of this research project.

Table 7.2.: *Completion Table of the Systems and Software Quality Standard (ISO 25010)*

	Importance	Completion
Functional stability		
Completeness	Very important	Features listed on the user stories defined in subchapter 3.4.2 are implemented
Correctness	Very important	Thorough tests on the `ValidationEngine` class and its components
Appropriateness	Important	Provided an overview of a validation process, and the `ValidationResult` model can be further extended to contain additional information
Reliability		
Maturity	Important	The system handles exception well and doesn't break when any error occurs
Availability	Very important	The system is runnable via a single command, and a live environment is also available
Fault tolerance	Important	The system can still run without the RabbitMQ instance and Redis by providing fallback options (e.g. `InMemoryStore` as a data store if Redis is not available)
Recoverability	Very important	Only if Redis is available, otherwise not complete
Performance efficiency		
Time behavior	Important	Running a validation process as an asynchronous operation to prevent long response time
Resource Utilization	Not important	Not complete
Capacity	Not important	Not complete
Usability		
Appropriateness Recognizability	Not important	Not complete

Learnability	Not important	Not complete
Operability	Very important	Provided a UI as an interface to interact with the FDS and `ValidationEngine`
User Error Protection	Somewhat important	Provided autocomplete input on the UI to prevent invalid JSONPath expression
User Interface Aesthetics	Very important	Provided a simple UI and a consistent design across all views
Accessibility	Not important	Not complete
Security		
Confidentiality	Somewhat important	Not complete
Integrity	Somewhat important	Not complete
Non-repudiation	Somewhat important	Not complete
Accountability	Not important	Not complete
Authenticity	Not important	Not complete
Compatibility		
Co-existence	Somewhat important	All components of the system works well with docker compose
Interoperability	Very important	The FDS shares data to the remaining components of the system (UI displays data provided by the FDS and RabbitMQ forwards the message published by the FDS)
Maintainability		
Modularity	Important	Thorough testing on the public interface of a component and not its implementation
Reusability	Not important	Created several utility functions to be used in different components
Analysability	Somewhat important	Not complete
Modifiability	Very important	Thorough integration test to make sure any modification won't break the specification of the system as a whole
Testability	Very important	Write unit tests before implementing the feature and automatically run the tests on every new merge request
Portability		
Adaptability	Not important	Used Docker, so that the application can be run on every operating system
Installability	Important	Implemented necessary steps in the Dockerfile for an easier build and install step
Replaceability	Not important	Not complete

8. Conclusion

In this chapter, the results and findings from previous chapters are summarized and an outlook on future improvements of the system is discussed.

8.1. Summary

Fraud detection and prevention plays a crucial role in building a profitable business by reducing the risk of damage done by any fraudulent activity. Furthermore, for organizations that utilize the autonomous teams structure, establishing a robust system for collaboration is important, so that the teams can work collectively on the primary goal of the organization.

This research project aims to introduce a system that enables independent teams to collaborate in a fraud detection process, by creating validation rules as their contribution. Each team could create validation rules to validate certain characteristics of a customer, based on their domain knowledge and views on how a fraudulent customer might be. All validation rules created by different teams will be evaluated by the system and as the result, the system outputs a probability of the corresponding customer being a fraud, by combining the result of every validation rule evaluation into a single numerical value, ranging from zero to one.

The system publishes each validation result into an exchange of a messaging system, that will broadcast the message into every message consumer bound to the exchange. Additional independent systems can be implemented as a message consumer to consume the message from the exchange and run actions accordingly. A UI is also provided as a graphical interface to manage validation rules and display the progress of a validation process in real time.

8.2. Outlook

The system built in this research project is far from perfect. Systems are living products that can and should be improved over time. One of the possible major improvement of the system is the application of authentication on the system. By implementing some sort of authentication mechanism, the security aspect of the system can be hugely improved, and any unauthorized use can be prevented.

Furthermore, it might be sometimes useful to group the validation rules, and run the rule evaluations only for certain groups. By assigning each rule into a group, there can be multiple types of validation processes, which can be modified based on the needs and requirements of the particular use case.

Last but not least, a solution to make sure that the data of the customer is safe and compliant with the current GDPR rules is definitely needed.

Bibliography

[1] Richard Bolton and David Hand. "Unsupervised Profiling Methods for Fraud Detection". In: *Conference on Credit Scoring and Credit Control* 7 (Sept. 2001).

[2] Jeff Friesen. "Extracting JSON Values with JsonPath". In: *Java XML and JSON: Document Processing for Java SE*. Berkeley, CA: Apress, 2019, pp. 299–322. ISBN: 978-1-4842-4330-5. DOI: 10.1007/978-1-4842-4330-5_10. URL: https://doi.org/10.1007/978-1-4842-4330-5_10.

[3] Erich Gamma et al. *Design Patterns*. Addison-Wesley, 1995.

[4] David Garlan. "Software architecture". In: *Proceedings of the conference on The future of Software engineering - ICSE '00* (2000). DOI: 10.1145/336512.336537.

[5] A. Harrison. *The Law of Athens: Procedure: 2*. Hackett Publishing Company, Inc., 1998.

[6] ISO. *ISO/IEC 25010:2011*. Online: https://www.iso.org/standard/35733.html; latest access: 01 VII 22. Mar. 2011.

[7] Yufeng Kou et al. "Survey of fraud detection techniques". In: *IEEE International Conference on Networking, Sensing and Control, 2004*. Vol. 2. 2004, 749–754 Vol.2. DOI: 10.1109/ICNSC.2004.1297040.

[8] Alexey Melnikov and Ian Fette. *The WebSocket Protocol*. RFC 6455. Online: http://www.rfc-editor.org/rfc/rfc6455.txt; latest access: 01 VII 22. RFC Editor, 2011.

[9] *New Data Shows FTC Received 2.8 Million Fraud Reports from Consumers in 2021*. Online: https://www.ftc.gov/news-events/news/press-releases/2022/02/new-data-shows-ftc-received-28-million-fraud-reports-consumers-2021-0; latest access: 01 VII 22. Feb. 2022.

[10] Henrik Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. Tech. rep. 2616. Online: https://www.rfc-editor.org/info/rfc2616; latest access: 01 VII 22. June 1999. 176 pp. DOI: 10.17487/RFC2616.

[11] Den Odell. "Build Tools and Automation". In: *Pro JavaScript Development: Coding, Capabilities, and Tooling*. Berkeley, CA: Apress, 2014, pp. 391–422. ISBN: 978-1-4302-6269-5. DOI: 10.1007/978-1-4302-6269-5_15. URL: https://doi.org/10.1007/978-1-4302-6269-5_15.

[12] Prisma. *prisma/Dockerfile at main*. Online: https://github.com/prisma/prisma/blob/main/docker/mongodb_replica/Dockerfile; latest access: 08 VII 22; commit: e5dc7629f567f1c6d2ccd2098aa7261326157ba5.

[13] Wendy Roome and Y. Richard Yang. *Application-Layer Traffic Optimization (ALTO) Incremental Updates Using Server-Sent Events (SSE)*. RFC 8895. Online: https://www.rfc-editor.org/info/rfc8895; latest access: 01 VII 22. Nov. 2020. DOI: 10.17487/RFC8895.

[14] Alan Jay Smith. "Cache Memories". In: *ACM Computing Surveys* 14.3 (1982), pp. 473–530. DOI: 10.1145/356887.356892.

[15] Hari Subramoni et al. "Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand". In: *2008 Workshop on High Performance Computational Finance*. 2008, pp. 1–8. DOI: 10.1109/WHPCF.2008.4745404.

[16] *Using Prisma with MongoDB*. Online: https://www.prisma.io/docs/guides/database/using-prisma-with-mongodb; latest access: 08 VII 22.

9. List of Abbreviations

AMQP	Advanced Message Queue Protocol
API	Application Programming Interface
B.C.	Before Christ
CI/CD	Continuous Integration / Continuous Delivery
CRUD	Create, Read, Update and Delete
DB	Database
FDS	Fraud Detection *Service*
GDPR	General Data Protection Regulation
HTTP	Hypertext Transfer Protocol
HW	Hardware
I/O	Input / Output
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
NPM	Node Package Manager
ORM	Object-Relational Mapping
REST	Representational State Transfer
SSE	Server-Sent Events
SW	Software
UI	User Interface
URL	Uniform Resource Locator
URI	Uniform Resource Identifier

A. Appendix

A.1. Supplemental Source Codes

```
model ValidationRule {
  id                   String    @id @default(auto()) @map("_id") @db.ObjectId
  name                 String    @unique
  skip                 Boolean
  priority             Int
  endpoint             String
  method               String
  failScore            Float
  condition            Json
  retryStrategy        Json?
  requestUrlParameter  Json?
  requestBody          Json?
  requestHeader        Json?
}
```

Listing A.1: Prisma *schema of a validation rule (Prisma)*

```
FROM mongo:4

ENTRYPOINT mongod --port $MONGO_REPLICA_PORT --replSet rs0 --bind_ip 0.0.0.0 & MONGOD_PID=$!; \
INIT_REPL_CMD="rs.initiate({ _id: 'rs0', members: [{ _id: 0, host: '$MONGO_REPLICA_HOST:$MONGO_REPLICA_PORT' }] })"; \
INIT_USER_CMD="db.createUser({ user: '$MONGO_INITDB_ROOT_USERNAME', pwd: '$MONGO_INITDB_ROOT_PASSWORD', roles: [ 'root' ] })"; \
until (mongo admin --port $MONGO_REPLICA_PORT --eval "$INIT_REPL_CMD && $INIT_USER_CMD"); do sleep 1; done; \
echo "REPLICA SET ONLINE"; wait $MONGOD_PID;
```

Listing A.2: *Dockerfile to run MongoDB as a replication set locally. Taken from Prisma GitHub repository (Docker)*

```yaml
version: "3.9"
services:
  fds:
    build: ./fds
    ports:
      - "8000:8000"
    environment:
      - ENABLE_CACHE: true
      - DATA_STORE: "redis"
      - DATABASE_URL: "mongodb://root:prisma@mongo:27018/tests?authSource=admin
        /"
      - REDIS: "redis://redis:6379/"
      - RABBITMQ_URL: "amqp://rabbit:5672"
      - RABBITMQ_MANAGEMENT_UI: "http://rabbit:15672"
    restart: on-failure
    depends_on:
      - redis
      - mongo
    #   - rabbit
    networks:
      - app-network
  ui:
    build: ./ui
    ports:
      - "3000:3000"
    depends_on:
      - fds
    networks:
      - app-network
  mongo:
    build: ./mongodb_replica
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: prisma
      MONGO_REPLICA_HOST: host.docker.internal
      MONGO_REPLICA_PORT: 27018
    ports:
      - '27018:27018'
    networks:
      - app-network
    volumes:
      - data-volume:/data/db
  redis:
    image: redis/redis-stack
    ports:
      - "6379:6379"
    networks:
      - app-network
  rabbit:
    image: rabbitmq:3-management-alpine
    ports:
      - "5672:5672"
      - "15672:15672"
    networks:
      - app-network
volumes:
  data-volume:
    driver: local
networks:
  app-network:
      driver: bridge
```

Listing A.3: *Docker Compose usage (YAML)*

A.2. Supplemental Figures

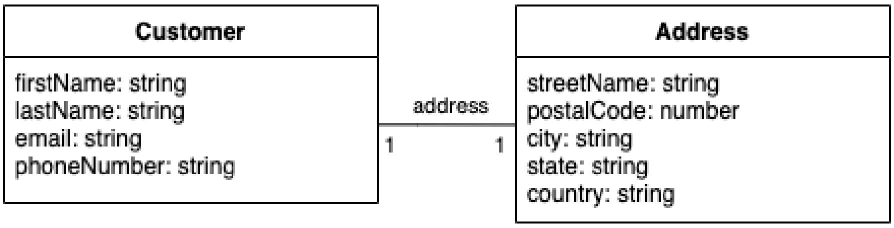

Figure A.1.: *UML diagram of the customer model*

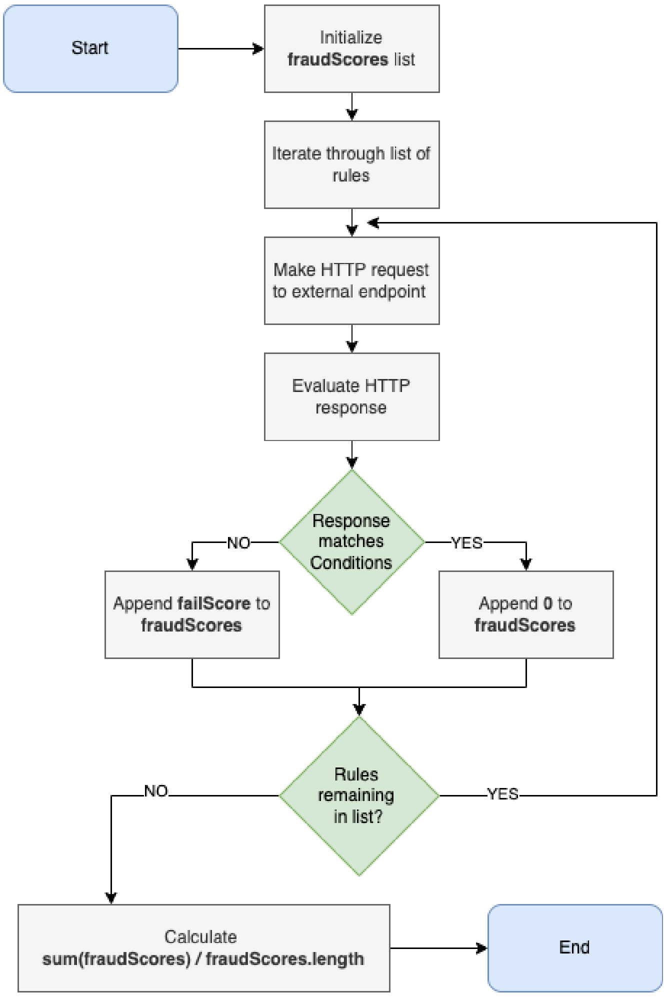

Figure A.2.: *Flow diagram of a validation process*

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Datum, Ort, Unterschrift