



**Hochschule für Technik  
und Wirtschaft Berlin**

University of Applied Sciences

*Design & Implementation of a Fraud Detection System for Autonomous Teams (Total  
pages should be: 50 [without Attachments])*

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 Mon Jun 20 11:37:38 UTC 2022

## **Abstract**

[Summary of the thesis]

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Background and Motivation . . . . .	3
1.2. Goal . . . . .	3
1.3. Scope . . . . .	3
<b>2. Fundamentals</b>	<b>4</b>
2.1. Kontext . . . . .	4
2.1.1. Domain . . . . .	4
2.1.2. Technologien . . . . .	4
2.1.3. Methoden und Konzepte . . . . .	4
2.2. ... . . . .	4
2.2.1. ... . . . .	4
2.2.2. ... . . . .	4
<b>3. Requirement Analysis (!)</b>	<b>5</b>
3.1. Goal? . . . . .	5
3.2. Application Environment . . . . .	5
3.3. Analysis / State of Art . . . . .	5
3.4. Requirements . . . . .	5
<b>4. Conception and Design</b>	<b>6</b>
4.1. System Architecture . . . . .	6
4.2. Software Architecture . . . . .	7
4.3. Technologies . . . . .	8
4.3.1. User Interface . . . . .	9
4.3.2. FDS . . . . .	10
4.3.3. Message Broker / Notification System . . . . .	10
4.3.4. Database and Caching Memory . . . . .	10
4.4. Software Design . . . . .	10
4.4.1. Controller . . . . .	11
4.4.2. Model . . . . .	16
4.4.3. View . . . . .	21
<b>5. Implementation</b>	<b>26</b>
5.1. Technologies and Architecture . . . . .	26
5.1.1. User Interface . . . . .	26
5.1.2. FDS . . . . .	27
5.1.3. RabbitMQ . . . . .	30
5.1.4. Redis . . . . .	31
5.1.5. MongoDB . . . . .	31

## Contents

5.1.6. Docker Compose . . . . .	31
5.2. Model . . . . .	31
5.2.1. Validation Rule . . . . .	32
5.2.2. Validation Result . . . . .	33
5.3. Controller . . . . .	34
5.3.1. Customer Validation on a Registration Event . . . . .	34
5.3.2. Validation Process . . . . .	35
5.3.3. Notification on Suspicious Cases . . . . .	39
5.3.4. Database Connection . . . . .	41
5.3.5. Validation Real Time Progress . . . . .	41
5.4. View . . . . .	43
<b>6. Test</b>	<b>44</b>
<b>7. Demonstration and Evaluation</b>	<b>45</b>
7.1. Ausblick . . . . .	46
<b>Bibliography</b>	<b>47</b>
<b>8. List of Abbreviations</b>	<b>49</b>
<b>9. Glossary</b>	<b>50</b>
<b>A. Appendix</b>	<b>I</b>
A.1. Supplemental Figures . . . . .	I
A.2. Supplemental Source Codes . . . . .	I
A.3. Quell-Code . . . . .	I
A.4. Tipps zum Schreiben Ihrer Abschlussarbeit . . . . .	I

# List of Figures

1.1. Example image: Who is Steinlaus?; Bildquelle [9] . . . . .	2
1.2. Beispielgrafik: Fressende Steinlaus; Bildquelle [8] . . . . .	2
4.1. System architecture diagram . . . . .	6
4.2. Software architecture diagram . . . . .	7
4.3. Software architecture diagram with technologies used . . . . .	9
4.4. System sequence diagram for a customer validation when a new customer is registered . . . . .	11
4.5. System sequence diagram for notifications on suspicious cases . . . . .	13
4.6. System sequence diagram for validation rules management . . . . .	14
4.7. System sequence diagram for validation rules management . . . . .	15
4.8. UML diagram of the validation rule model . . . . .	17
4.9. UML diagram of the validation result model . . . . .	20
4.10. Mock up of the rule management form page . . . . .	23
4.11. Mock up of the validation form page . . . . .	24
4.12. Mock up of the validation progress page . . . . .	24
5.1. Flow diagram of a validation process . . . . .	36
A.1. UML diagram of the customer model . . . . .	I

# List of Tables

1.1. Übersicht: Untersuchte Steinläuse . . . . .	2
--	---

# Listings

4.1. Validation rule example (JSON) . . . . .	18
4.2. Validation rule <b>condition</b> attribute example with ALL condition (JSON) . . . . .	18
4.3. Validation rule <b>condition</b> attribute example with ANY condition (JSON) . . . . .	19
4.4. Validation result example (JSON) . . . . .	20
5.1. Creating a new Vite project (Shell) . . . . .	26
5.2. Creating a new Node.JS program (Shell) . . . . .	28
5.3. Installing and configuring TypeScript compiler (Shell) . . . . .	28
5.4. Installing Express.JS (Shell) . . . . .	28
5.5. Installing Prisma (Shell) . . . . .	28
5.6. Set up project with Prisma . . . . .	28
5.7. Configuring database type and URL (Prisma) . . . . .	28
5.8. Update Prisma client API to include generated interfaces from the schema (Shell) . . . . .	29
5.9. Dockerfile for FDS (Docker) . . . . .	30
5.10. Running a RabbitMQ instance with Docker (Shell) . . . . .	30
5.11. Running a Redis instance with Docker (Shell) . . . . .	31
5.12. Running a MongoDB instance with Docker (Shell) . . . . .	31
5.13. TypeScript interface of a validation rule (TypeScript) . . . . .	32
5.14. Establishing database connection with Prisma (TypeScript) . . . . .	33
5.15. Example usage of Prisma (TypeScript) . . . . .	33
5.16. TypeScript interface of a validation result (TypeScript) . . . . .	34
5.17. ValidationEngine class builder pattern using method chaining (TypeScript) . . . . .	35
5.18. Accessing runtime information using JSONPath expression (TypeScript) . . . . .	36
5.19. Usage of the singleton pattern and dependency injection in Agent class (TypeScript) . . . . .	37
5.20. NumberOperator example (TypeScript) . . . . .	38
5.21. The usage of OperatorFactory class in the Evaluator class (TypeScript) . . . . .	39
5.22. EvaluatorFactory usage in ValidationEngine class (TypeScript) . . . . .	39
5.23. Opening a connection to RabbitMQ instance (TypeScript) . . . . .	40
5.24. Publishing a validation result to the RabbitMQ exchange (TypeScript) . . . . .	40
5.25. Consuming a message published to the RabbitMQ exchange (TypeScript) . . . . .	41
5.26. Publishing events to the EventEmitter on certain validation events (TypeScript) . . . . .	42
5.27. Writing to SSE stream when certain events are published (TypeScript) . . . . .	42
A.1. <i>Prisma</i> schema of a validation rule (Prisma) . . . . .	I

# 1. Introduction

Vorliegendes Template enthält exemplarisch (und damit unvollständig) Gliederungspunkte, Bestandteile und Hinweise für ein typisches Softwareentwicklungsprojekt, bei dem ein Prototyp erstellt wird. Es dient als Hilfestellung zu Ihrer weiteren Verwendung. Selbstverständlich müssen Sie selbst weitere Ergänzungen und Anpassungen vornehmen.

**Viel Erfolg sowie gutes Gelingen bei Ihrer Abschlussarbeit!**

Der Textteil beginnt hier und wird arabisch mit dieser Seite beginnend mit »1« arabisch nummeriert. Der Textteil gliedert sich in Kapitel und Unterkapitel. Soll jede Hierarchieebene benannt werden, dann ist folgende Terminologie üblich:

- 1. Hierarchieebene: Chapter
- 2. Hierarchieebene: Section
- 3. Hierarchieebene: Subsection
- 4. Hierarchieebene: Subsubsection

Der inhaltliche Aufbau einer Abschlussarbeit im Studiengang *Angewandte Informatik* hängt selbstverständlich vom Thema und vom Inhalt ab. Abweichungen von der diesem Template zu Grunde liegenden Gliederungsstruktur sind immer möglich, manchmal sogar zwingend notwendig. Stimmen Sie sich diesbezüglich immer mit Ihren Gutachter(inne)n ab.

Vergessen Sie niemals, all Ihre verwendeten Quellen anzugeben und korrekt zu zitieren<sup>1</sup>. Quellen können manuell referenziert und im Quellenverzeichnis eingetragen werden. Ergänzend bieten viele Textverarbeitungssysteme auch ausgelagerte Quellenverwaltungsdateien und -systeme an, über die mittels entsprechender Befehle im Textteil zitiert werden kann<sup>2</sup>.

Visualisieren Sie im Textteil angemessen, z.B. mittels Abbildungen und Tabellen. Vorliegendes Template enthält beispielhaft eingebundene Abbildungen und eine Tabelle (vgl. f.), welche der Steinlausforschung<sup>3</sup> entnommen sind.

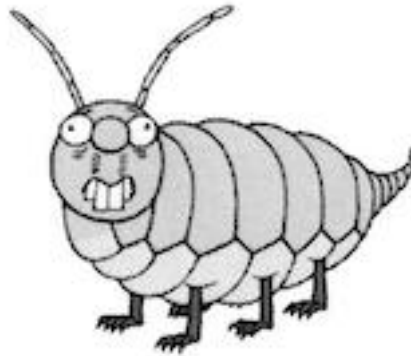
<sup>1</sup>Ergänzende Informationen können Sie auch in eine Fußnote auslagern. Hier wird die Fußnote dazu genutzt, um Ihnen bei Interesse am Thema Zitation vertiefende Quellen (z.B. [1] oder [2]) anzubieten.

<sup>2</sup>Wie Sie hoffentlich feststellen werden, erfolgt die Literaturverwaltung in diesem Template mittels einer \*.bib-Datei (diese enthält die verwendeten Quellen), welche die \*.tex-Datei mittels Verwendung von biblatex und bibtex ergänzt.

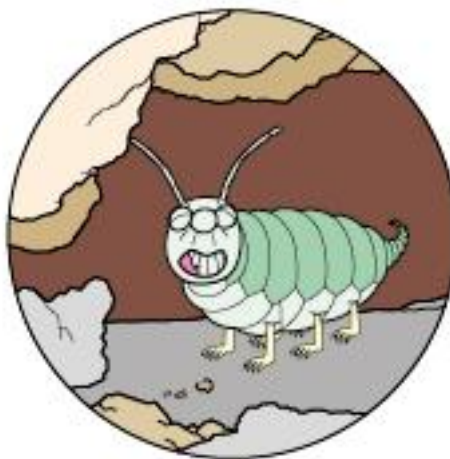
<sup>3</sup>Analog zu Straube (In: [13]) handelt es sich bei der Steinlaus (*petrophaga lorioti*) um das »kleinste einheimische Nagetier«. Als stimmungsaufhellender Endoparasit erreicht es eine Größe von ca. 0,3 bis 3 mm und stammt aus der Familie der Lapivora. Die Steinlaus kommt ubiquitär vor und ist in der Regel apathogen.



## 1. Introduction



**Figure 1.1.:** Example image: Who is Steinlaus?; Bildquelle [9]



**Figure 1.2.:** Beispielgrafik: Fressende Steinlaus; Bildquelle [8]

**Table 1.1.:** Übersicht: Untersuchte Steinläuse

Untersuchte Objekte mit Lokation des Habitats		
ID (nickname)	Ort	Grö"se/Länge (in mm)
1 (Rosalinde)	Berlin, Mauerpark	1.4
2 (Devil in disguise)	Brandenburg, BER-Airport	2.8
3 (Hannes)	Berlin, Olympia-Stadion	2.1
4 (Her Majesty)	Berlin, Humboldt-Forum	2.0

## *1. Introduction*

### **1.1. Background and Motivation**

The background and motivation of the thesis is to get my bachelor degree.

### **1.2. Goal**

Goal of the thesis is to build a cool software

### **1.3. Scope**

Scope of the thesis is to not build a new internet protocol

## 2. Fundamentals

**TODO!** [Beschreibung des Kontextes der Arbeit mit allen durch die Problemstellung tangierten Bereichen, Methoden, Theorien, Erkenntnissen, Technologien, ... ]

### 2.1. Kontext

#### 2.1.1. Domain

#### 2.1.2. Technologien

#### 2.1.3. Methoden und Konzepte

### 2.2. ...

#### 2.2.1. ...

#### 2.2.2. ...

## 3. Requirement Analysis (!)

[Beschreibung der Erhebung, Granularisierung und Priorisierung der zu Grunde liegenden Anforderungen]

### 3.1. Goal?

Hi mom, I'm . trying to cite ISO thingy [6]

### 3.2. Application Environment

### 3.3. Analysis / State of Art

### 3.4. Requirements

Optionals:

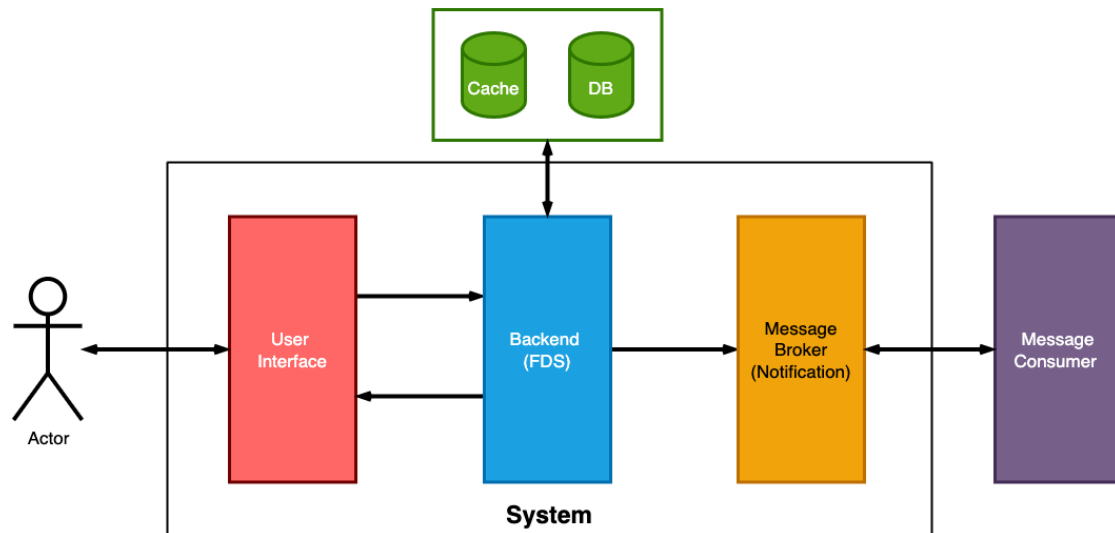
- *Rahmenbedingung*
- Methods

## 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<sup>1</sup> 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.

<sup>1</sup>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 [15].

## 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.”[5] A software architecture diagram is therefore created to help visualize the structure, functions and role of each component of the system.

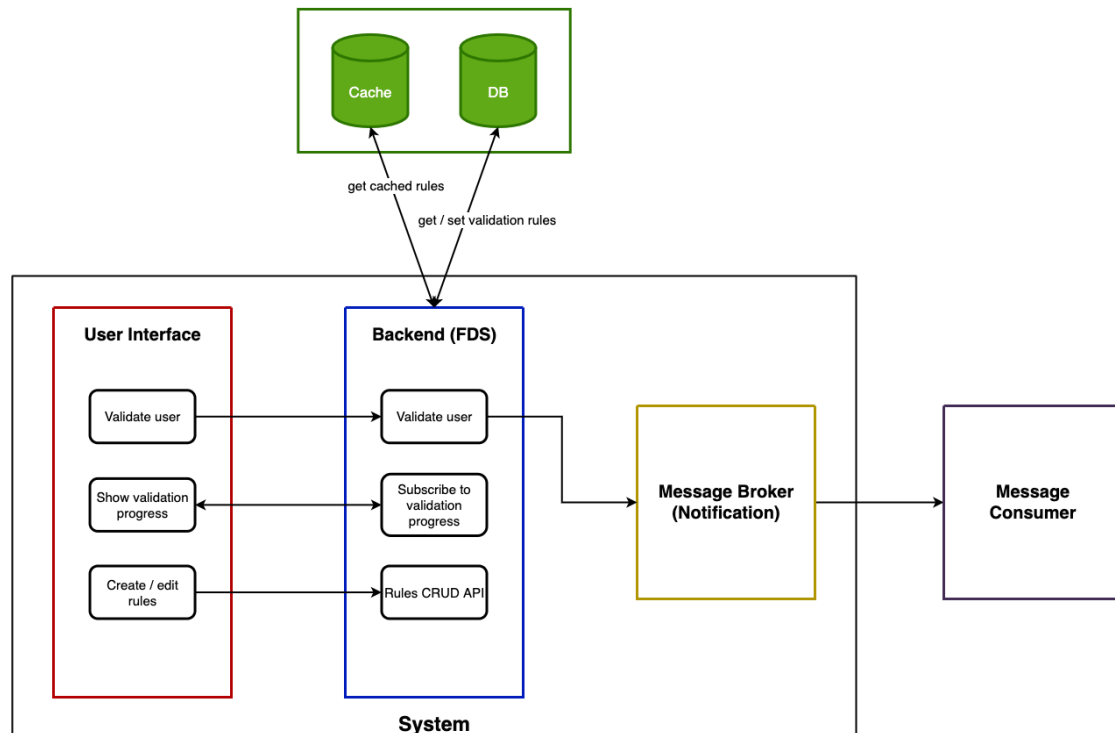


Figure 4.2.: Software architecture diagram

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 customer service agents, 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

#### 4. Conception and Design

be discussed in subsection 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 payload<sup>2</sup>. 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<sup>3</sup> 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* [4, 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

---

<sup>2</sup>A request payload refers to data sent by a client to the server during an HTTP request, usually attached to the request body[11, section "4.3 Message Body"].

<sup>3</sup>In [10], 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.

## 4. Conception and Design

- Database and caching memory

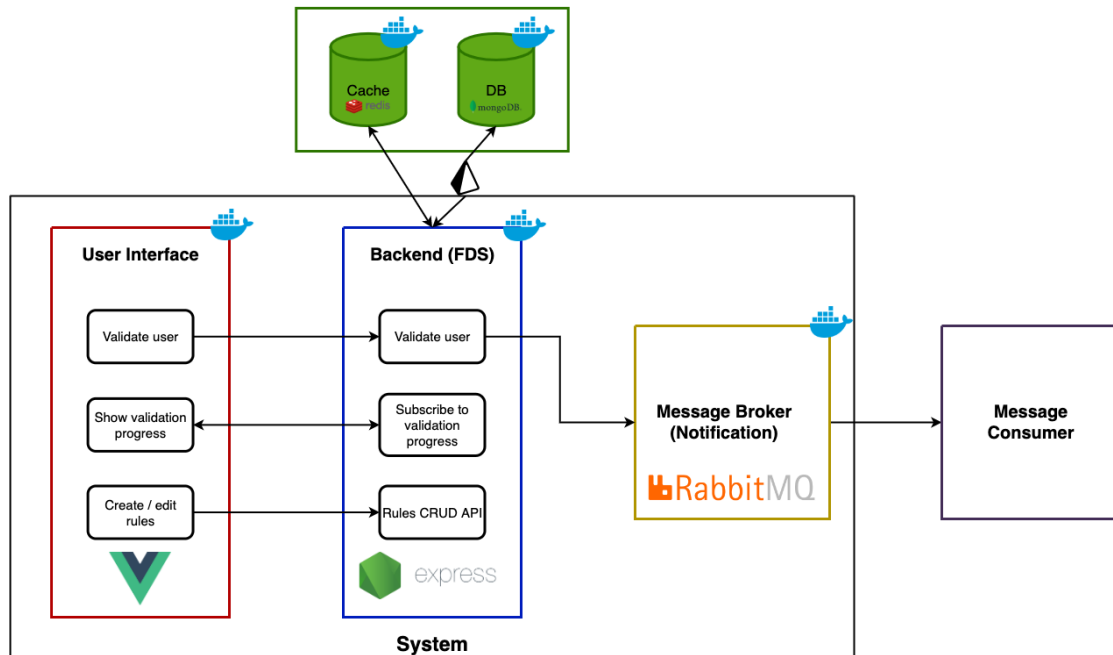


Figure 4.3.: Software architecture diagram with technologies used

The previous software architecture diagram is therefore extended with additional logos of the technology used for each component of the system. All internal components of the system should be run as a Docker<sup>4</sup> 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<sup>5</sup>.

### 4.3.1. User Interface

The technology used to build the user interface is VueJS (3. Version, also known as Vue3)<sup>6</sup>, 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<sup>7</sup>, rather than plain JavaScript, which is also supported by Vue3.

<sup>4</sup>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/>.

<sup>5</sup>Docker Compose is an open source tool for running multiple Docker containers. GitHub repository: <https://github.com/docker/compose>.

<sup>6</sup>Vue3 is an open source JavaScript framework to build user interfaces. GitHub repository: <https://github.com/vuejs/core>.

<sup>7</sup>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. Conception and Design

### 4.3.2. FDS

The technology chosen to build the FDS is Node.JS<sup>8</sup>. 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<sup>9</sup> 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<sup>10</sup>, 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 result actually reaches the consumers. The technology chosen for this component is RabbitMQ<sup>11</sup>, 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<sup>12</sup> will be used as the database system of choice. Redis<sup>13</sup> is chosen as the technology of choice for the caching memory because of its simple API and reliability.

## 4.4. Software Design

The system was implemented using the Model-View-Controller (MVC) programming approach. The system does not strictly adhere to the MVC pattern, but uses it as a guideline to categorize the components of the system and its functionalities. Krasner

---

<sup>8</sup>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>.

<sup>9</sup>Express JS is an open source web application framework for Node.JS. GitHub repository: <https://github.com/expressjs/expressjs.com>.

<sup>10</sup>Prisma is an open source ORM for Node.JS and TypeScript. GitHub repository: <https://github.com/prisma/prisma>.

<sup>11</sup>RabbitMQ is a messaging broker, enabling the distribution of messages across multiple clients. RabbitMQ homepage: <https://www.rabbitmq.com/>.

<sup>12</sup>MongoDB is a source-available NoSQL database developed by MongoDB Inc. MongoDB homepage: <https://www.mongodb.com/>.

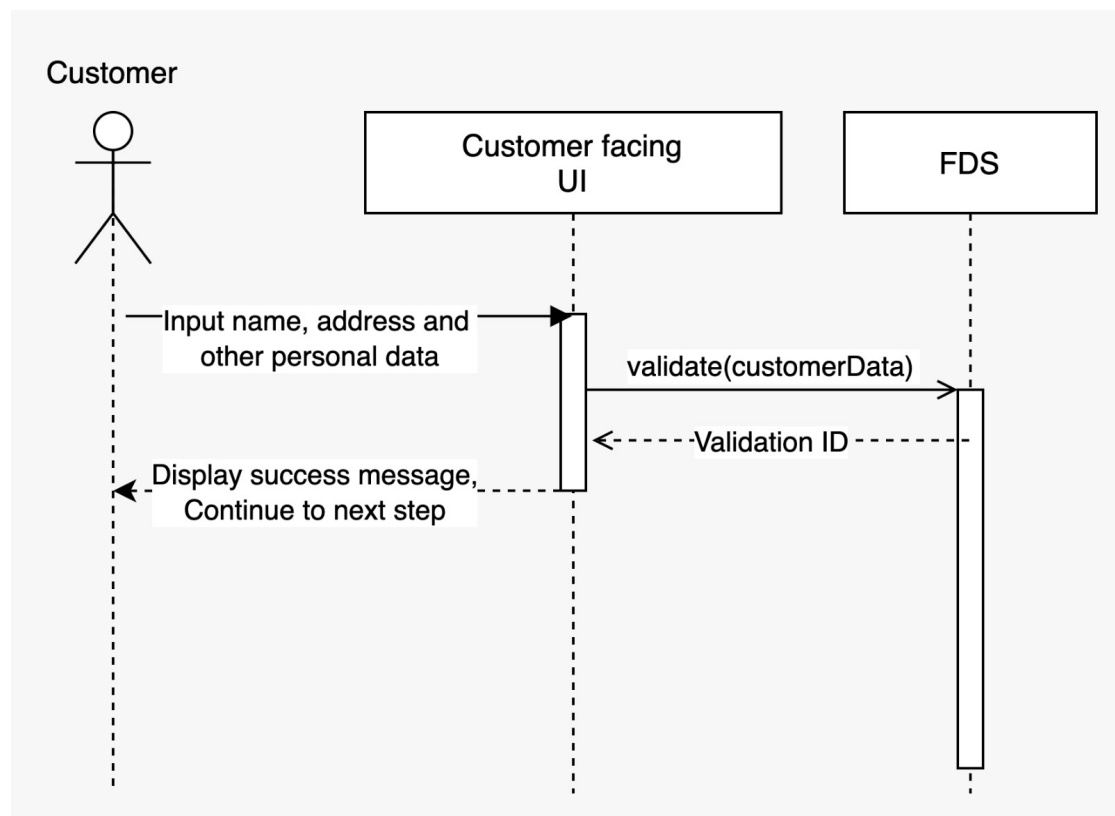
<sup>13</sup>Redis is an open-source in-memory data structure store. GitHub repository: <https://github.com/redis/redis>.

and Pope [7] discussed the benefit of using the MVC pattern to provide modularity by isolating functional components of the system, making it easier to design and modify.

### 4.4.1. Controller

Krasner and Pope defined an application's controller in [7] as an interface to associate the components of the system (*model and view*) and incoming events (*such as event coming from input devices*). In other words, the controller component is responsible in defining the flow or sequence of a system based on an incoming event, routing commands to the appropriate model and updating the view in the process. In this subsection, an analysis of the use cases listed on **TODO: ref to use cases** will be done, and consequentially a sequence of operations will be defined for each specific use case. An additional sequence will also be defined as a way 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”

#### 4. Conception and Design

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

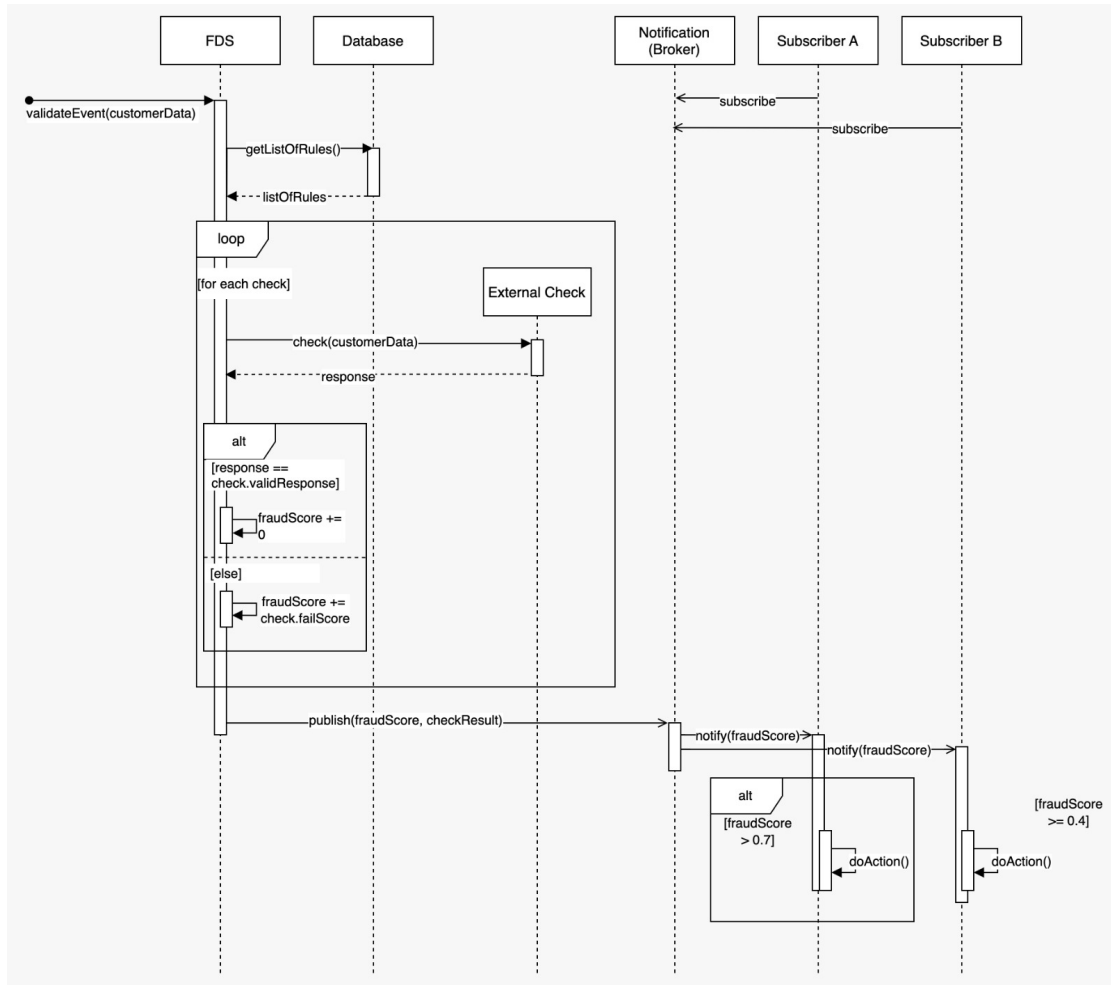
To fulfill the requirements listed on **TODO: Link Analysis**, a further examination of the following use case should be done:

"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

#### 4. Conception and Design



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

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<sup>14</sup>

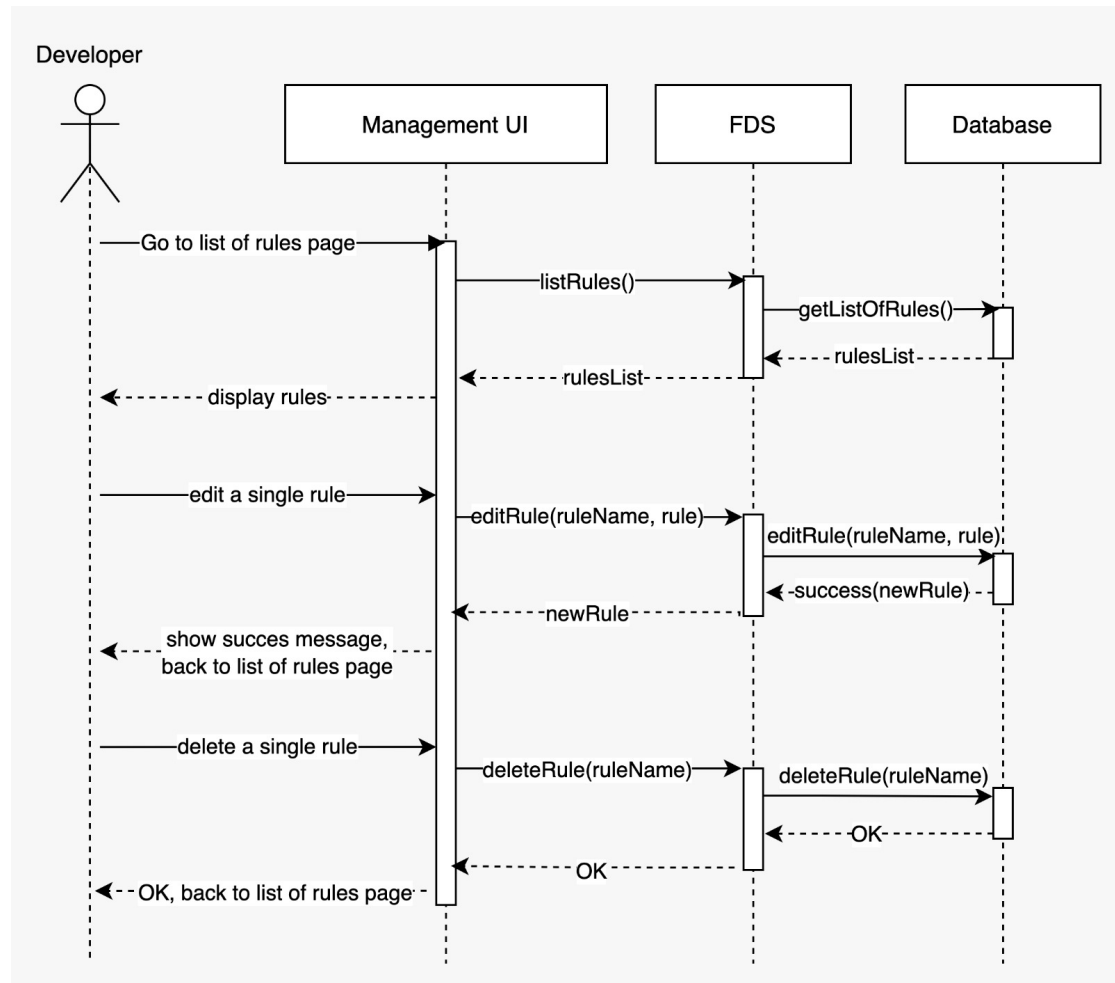
#### Managing Validation Rules

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

<sup>14</sup>For example: sending an email notification if the fraud score exceeds 0.7.

#### 4. Conception and Design



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

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:

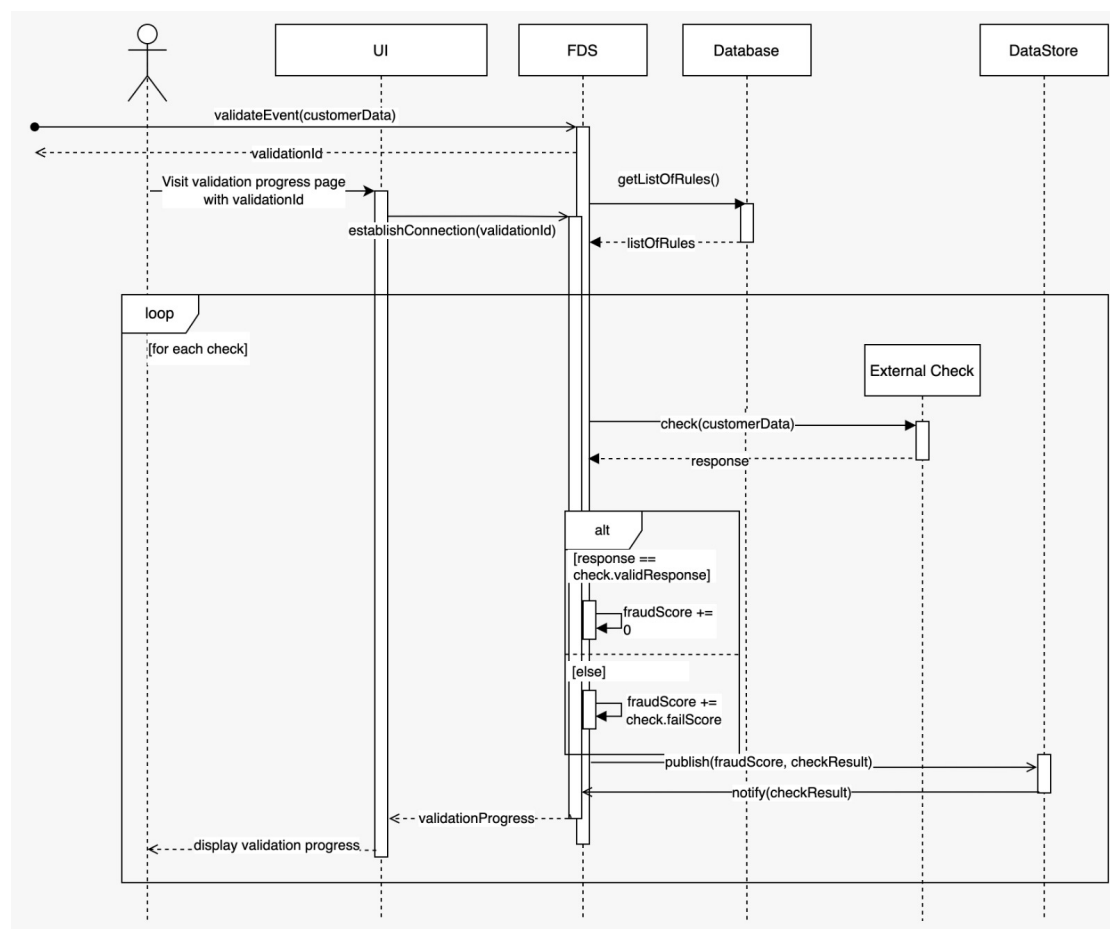
- A user (e.g. Developer) can access the management UI and go to the page that displays a list of available validation rules
- The FDS retrieves a list of validation rules from the database
- User can click on a single rule and edit the rule
- The management UI makes an HTTP PUT<sup>15</sup> request 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

<sup>15</sup>In [11, "9.6 PUT"], HTTP PUT method is described as a method to store or modify an entity, defined by the Request-URI.

#### 4. Conception and Design

- 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<sup>16</sup> 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<sup>17</sup> 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



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

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.

<sup>16</sup>In [11, “9.7 DELETE”], HTTP DELETE method is described as a method to delete a resource on the host server, pointed by the Request-URI.

<sup>17</sup>In [11, “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.

#### 4. Conception and Design

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
- 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<sup>18</sup>
- 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. Model

As mentioned by Krasner and Pope [7], an application's model is a domain specific simulation or implementation of a structure. The model component of an MVC application contains business logic and manages the state of an application as well as its storage. The essential models of the system can be defined by revisiting the sequences listed on subsection 4.4.1 and identifying the important structures needed to fulfill the requirements needed.

##### 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 to be used to make the request

---

<sup>18</sup>A data store in this context can be a caching memory or a simple class to store some temporary information.

#### 4. Conception and Design

- Request header<sup>19</sup>
- 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 **TODO: Add retry strategy implementation**.

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

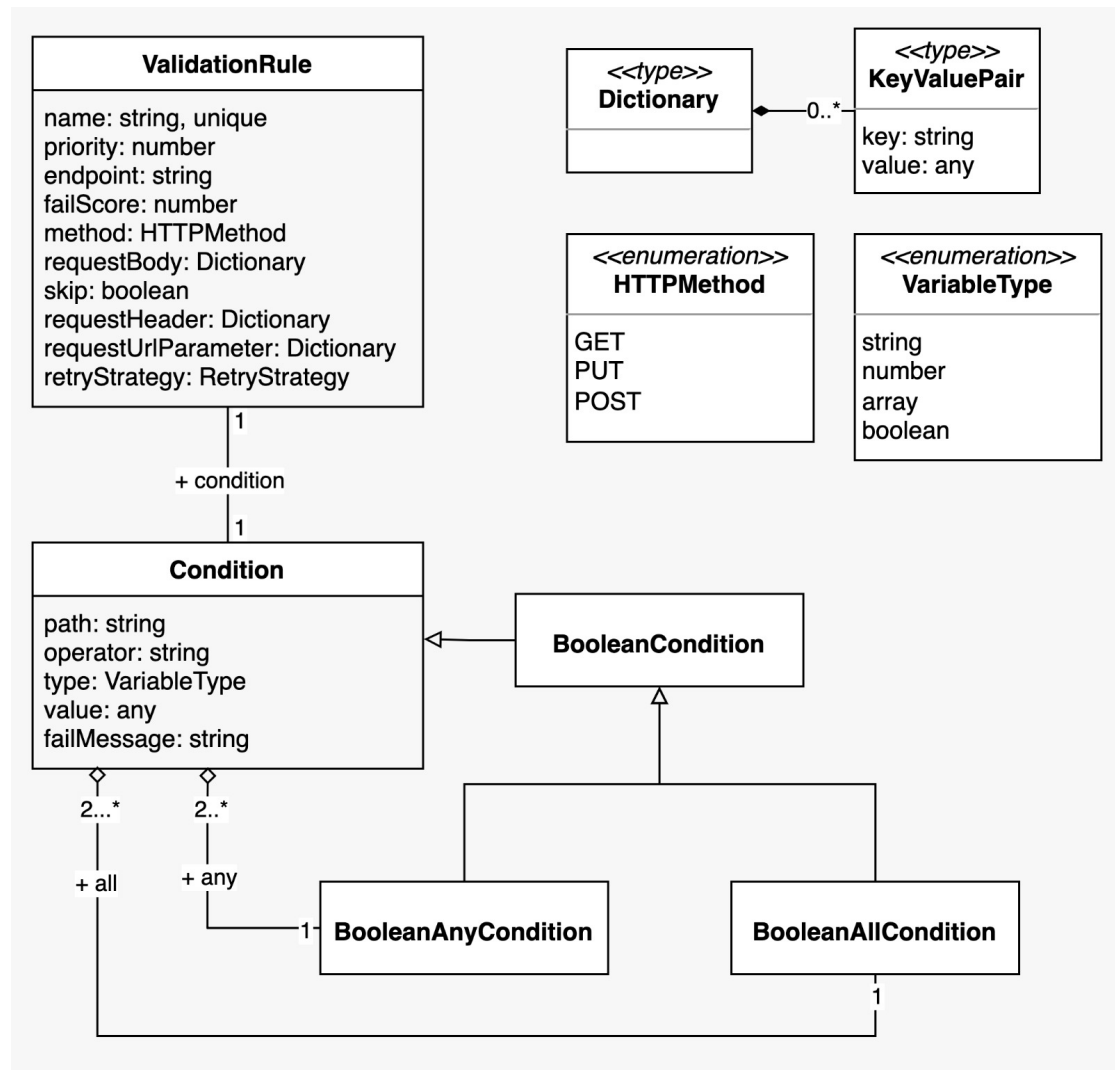


Figure 4.8.: UML diagram of the validation rule model

The *condition* attribute plays an important role for a validation rule, as it defines how the response returned by the external endpoint should be structure to pass a rule

<sup>19</sup>In [11, "5.3 Request Header Fields"]: request header is defined as additional information passed by the client to the host server about the particular request or about the client.



#### 4. Conception and Design

evaluation. It is intended to design the condition attribute to be robust and configurable. The *path* of a condition defines a JSONPath[3] 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 type of the attribute accessed by the *path* attribute. The *type* attribute determines which type of operators are available to use<sup>20</sup>. 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 **TODO: Add operator implementation**. 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.

```
1  {
2    "name": "Example",
3    "skip": false,
4    "priority": 2,
5    "endpoint": "http://localhost:8000/validate",
6    "method": "GET",
7    "failScore": 0.7,
8    "condition": {
9      "path": "$.response.statusCode",
10     "type": "number",
11     "operator": "eq",
12     "value": 200,
13     "failMessage": "Status code doesn't equal to 200"
14   },
15   "requestUrlParameter": {},
16   "requestBody": {},
17   "requestHeader": {}
18 }
```

**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 need to define whether 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.

```
1  {
2    "condition": {
3      "all": [
4        {
5          "path": "$.response.statusCode",
6          "type": "number",
7          "operator": "eq",
8          "value": 200,
9          "failMessage": "Status code doesn't equal to 200"
10        }
11      ]
12    }
13 }
```

<sup>20</sup>For example: a condition with *type* "string" cannot use the "incl" operator, because the "incl" operator is only available for "array" type.

#### 4. Conception and Design

```
11     {
12         "path": "$.response.body.valid_address",
13         "type": "boolean",
14         "operator": "eq",
15         "value": false,
16         "failMessage": "Address is invalid"
17     }
18 ]
19 }
20 }
```

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

```
1  {
2      "condition": {
3          "any": [
4              {
5                  "path": "$.response.statusCode",
6                  "type": "number",
7                  "operator": "eq",
8                  "value": 200,
9                  "failMessage": "Status code doesn't equal to 200"
10             },
11             {
12                 "path": "$.response.body.valid_address",
13                 "type": "boolean",
14                 "operator": "eq",
15                 "value": false,
16                 "failMessage": "Address is invalid"
17             }
18         ]
19     }
20 }
```

**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 **TODO: Link implementation**. 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, runned 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

#### 4. Conception and Design

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 **TODO: cite to implementation**).

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

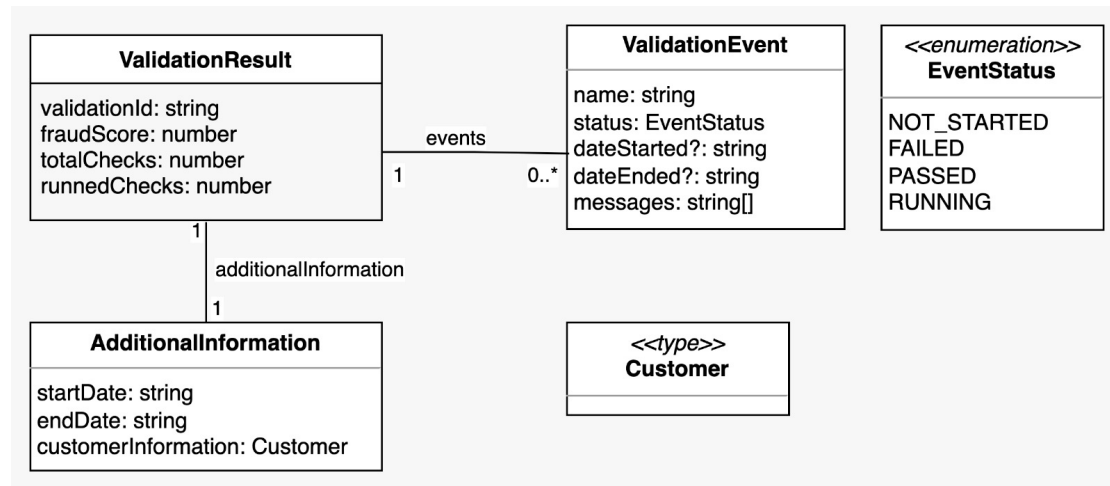


Figure 4.9.: UML diagram of the validation result model

```

1  {
2    "validationId": "3112dc4a-45f5-41b8-883a-715dcbe9a490",
3    "totalChecks": 3,
4    "runnedChecks": 2,
5    "skippedChecks": [
6      "Skip rule",
7    ],
8    "additionalInfo": {
9      "startDate": "2022-06-12T09:16:24.618Z",
10     "customerInformation": {
11       "firstName": "Scooby",
12       "lastName": "Doo",
13       "address": {
14         "streetName": "Suite 5000 185 Berry St",

```

## 4. Conception and Design

```
15     "city": "San Fransisco",
16     "state": "CA",
17     "country": "United States",
18     "postalCode": 94107
19   },
20   "email": "scooby-doo@fraud.co",
21   "phoneNumber": "123131123"
22 },
23 },
24 "events": [
25   {
26     "name": "User's email domain is not blacklisted",
27     "status": "PASSED",
28     "dateStarted": "2022-06-12T09:16:34.694Z",
29     "dateEnded": "2022-06-12T09:16:39.725Z",
30     "messages": []
31   },
32   {
33     "name": "User's email is not blacklisted",
34     "status": "FAILED",
35     "dateStarted": "2022-06-12T09:16:39.725Z",
36     "dateEnded": "2022-06-12T09:16:44.756Z",
37     "messages": [
38       "User's email is blacklisted!"
39     ]
40   }
41 ],
42 "fraudScore": 0.425
43 }
```

**Listing 4.4:** Validation result example (JSON)

### 4.4.3. View

In [7], Krasner and Pope described an application's view as the graphical representation of an application's model. Several mock-ups are made 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<sup>21</sup> 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.

<sup>21</sup>An understanding on how HTTP works is a prerequisite to use the form.

#### 4. Conception and Design

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 subsubsection 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<sup>22</sup>. 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 fields.

#### Validation Form

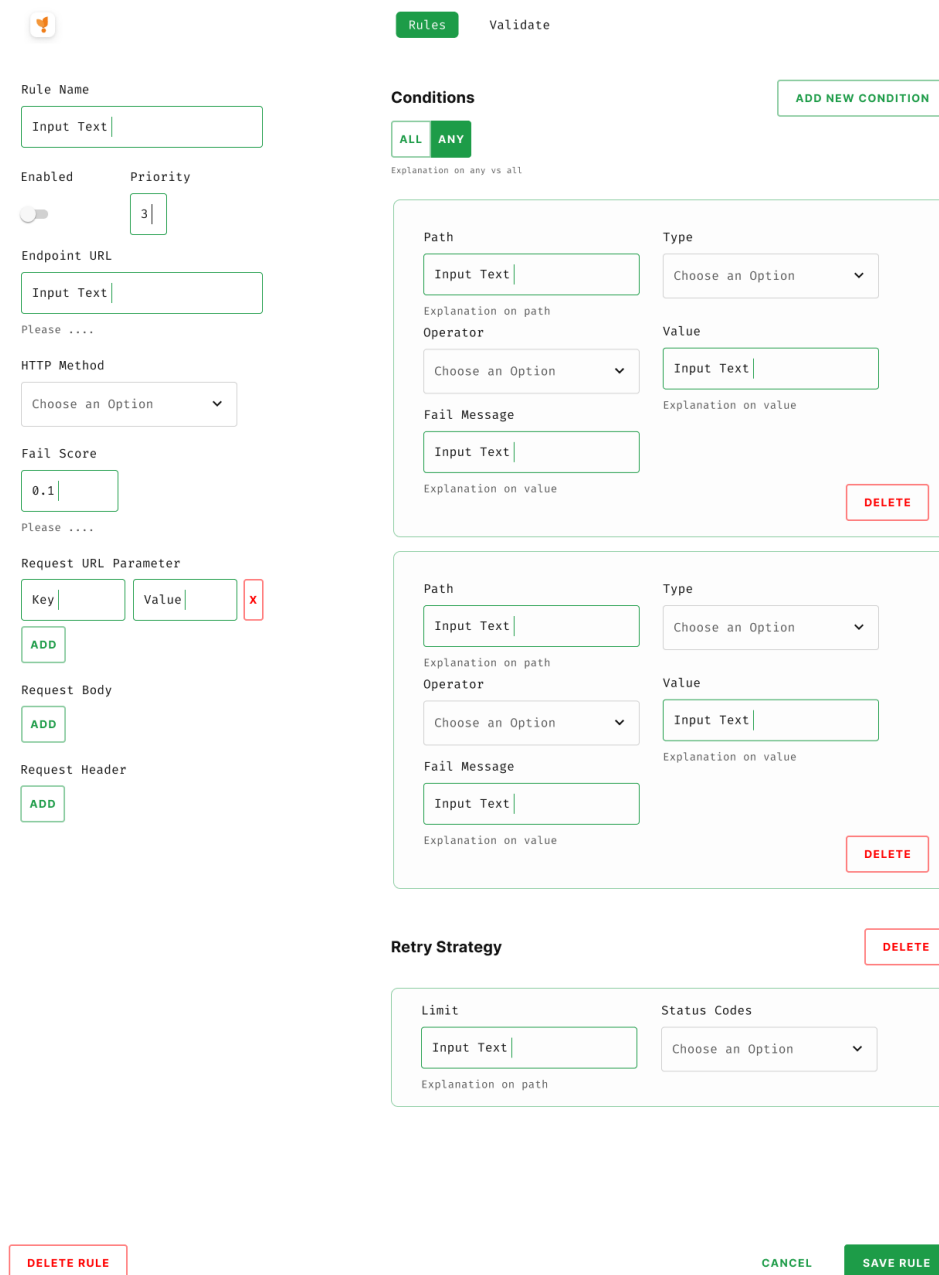
A sample registration form is also created to give the user a possibility to run a validation process on a certain customer. The validation form is intended to be used by end customer, as a mean to register his-/herself into the system. Internal employees can also use the validation form to test the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

---

<sup>22</sup>For example, on *type* field, user can only choose on of the following: (number, string, array, boolean).

## 4. Conception and Design



The mockup shows a rule management interface. On the left, a sidebar contains a logo and a list of rule types: Rules, Validate, and a third button. The main area is titled 'Rules' and 'Validate'. It features a form for creating or editing a rule. The form includes fields for Rule Name, Enabled (toggle), Priority (3), Endpoint URL, HTTP Method (dropdown), Fail Score (0.1), Request URL Parameter (Key, Value, X), Request Body, and Request Header. Below these are buttons for ADD, DELETE RULE, CANCEL, and SAVE RULE. The right side of the form is titled 'Conditions' and 'Retry Strategy'. It contains two sections: 'Conditions' and 'Retry Strategy'. The 'Conditions' section has a dropdown for ALL/ANY, a button for ADD NEW CONDITION, and two condition blocks. Each block has fields for Path, Type, Operator, Value, and Fail Message, with a DELETE button. The 'Retry Strategy' section has a dropdown for Limit, a dropdown for Status Codes, and a DELETE button.

Rule Name  
Input Text

Enabled ☐ Priority 3

Endpoint URL  
Input Text

Please ....

HTTP Method  
Choose an Option

Fail Score  
0.1

Please ....

Request URL Parameter  
Key Value X  
ADD

Request Body  
ADD

Request Header  
ADD

DELETE RULE

CANCEL SAVE RULE

Rules Validate

Conditions

ALL ANY

Explanation on any vs all

ADD NEW CONDITION

Path Type  
Input Text Choose an Option

Explanation on path

Operator Value  
Choose an Option Input Text

Fail Message Explanation on value  
Input Text

DELETE

Path Type  
Input Text Choose an Option

Explanation on path

Operator Value  
Choose an Option Input Text

Fail Message Explanation on value  
Input Text

DELETE

Retry Strategy

DELETE

Limit Status Codes  
Input Text Choose an Option

Explanation on path

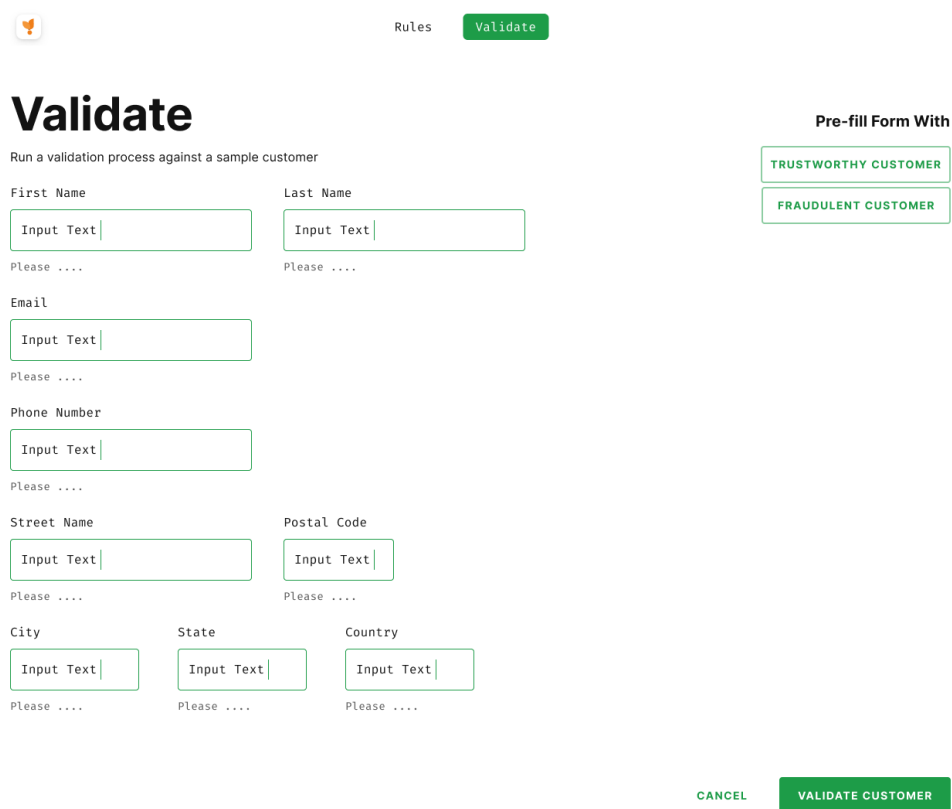
Figure 4.10.: Mock up of the rule management form page

### 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.

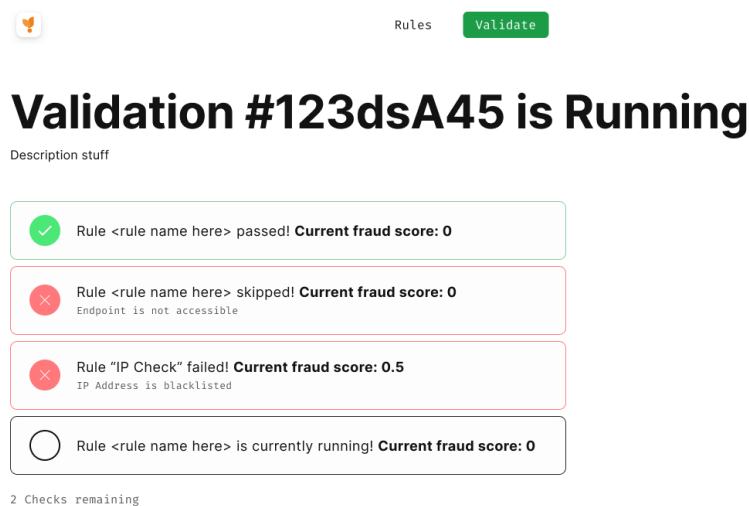
The page displays the current progress of a certain validation in real time. It

## 4. Conception and Design



The mock up shows a validation form page. At the top, there is a logo on the left, a 'Rules' link in the center, and a green 'Validate' button on the right. Below the header, the main heading is 'Validate' in a large, bold font. Underneath, a sub-header reads 'Run a validation process against a sample customer'. The form consists of several input fields: 'First Name', 'Last Name', 'Email', 'Phone Number', 'Street Name', 'Postal Code', 'City', 'State', and 'Country'. Each field has a placeholder text 'Please ...'. To the right of the form, there is a section titled 'Pre-fill Form With' containing two buttons: 'TRUSTWORTHY CUSTOMER' and 'FRAUDULENT CUSTOMER'. At the bottom right, there are two buttons: 'CANCEL' and 'VALIDATE CUSTOMER'.

Figure 4.11.: Mock up of the validation form page



The mock up shows a validation progress page. At the top, there is a logo on the left, a 'Rules' link in the center, and a green 'Validate' button on the right. Below the header, the main heading is 'Validation #123dsA45 is Running' in a large, bold font. Underneath, a sub-header reads 'Description stuff'. The progress section contains four items, each with a circular icon and text: 1. A green checkmark icon, 'Rule <rule name here> passed! Current fraud score: 0'. 2. A red 'X' icon, 'Rule <rule name here> skipped! Current fraud score: 0', with a sub-text 'Endpoint is not accessible'. 3. A red 'X' icon, 'Rule "IP Check" failed! Current fraud score: 0.5', with a sub-text 'IP Address is blacklisted'. 4. A grey circle icon, 'Rule <rule name here> is currently running! Current fraud score: 0'. At the bottom, it says '2 Checks remaining'.

Figure 4.12.: Mock up of the validation progress page

#### 4. *Conception and Design*

represents the *events* attribute of a validation result in a timeline, displaying the events in a list ordered by its *dateEnded* attribute. The page gives the user information regarding the evaluation result of each validation rule (success, failure or in progress), 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 design of the system is defined and the functionalities of each component of the system is clear, the implementation of the system can finally start. This chapter describes the detailed information on the implementation of the structure and functionalities listed in chapter 4.

### 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. The prerequisites that need to be met before setting up the projects are:

- Node.JS version 16 is installed
- NPM is installed
- Docker is installed
- Git is installed

For certain components of the system, a live environment is configured, so that the system is accessible via a URL, without having to set it up on a local machine.

#### 5.1.1. User Interface

As mentioned in subsection 4.3.1, the UI is a web application, built using Vue3 and TypeScript.

##### Setup

Nowadays, most front end projects use some kind of build tool to help build, test, and develop web applications<sup>1</sup>. The build tool *Vite*<sup>2</sup> is used to build, test and develop the UI. A new Vite project is created by running the following command in a shell terminal:

```
1 npm create vite@latest ui --template vue-ts
```

**Listing 5.1:** *Creating a new Vite project (Shell)*

A version control to track and manage the progress done in this particular project is also used. The version control used in this project is git and a code hosting platform for version control is also used for the project (GitHub).

---

<sup>1</sup>For detailed information on the role of a build tool in front end development, please take a look at [12].

<sup>2</sup>*Vite* is an open source front end build tool. GitHub repository: <https://github.com/vitejs/vite>.

## 5. Implementation

### Component Library

A component library is used in this project to speed up the development. The component library used in this project is *NaiveUI*<sup>3</sup>. 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.

### Docker

### TODO. Implementation not done

### 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 to enable this functionality is *GitHub actions*<sup>4</sup>. The configured CI/CD actions in this project are:

- Run test and build, when there's a new pull request (PR)<sup>5</sup> to main branch
- Run release and bump version of the app, when there's a new commit to main branch

### Deployment

A live environment is available for the UI. The live environment is made possible by using *Netlify*<sup>6</sup>. Every change made to the main branch will be built and deployed on the Netlify platform automatically.

#### 5.1.2. FDS

As described in subsection 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.

### Setup

To set up a new Node.JS project, run the following command in a shell terminal:

---

<sup>3</sup>*NaiveUI* is a Vue3 component library. GitHub repository: <https://github.com/TuSimple/naive-ui>.

<sup>4</sup>*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>.

<sup>5</sup>*Pull request (PR)* is a request from a developer to merge certain changes on a dedicated branch to the main branch of a repository.

<sup>6</sup>*Netlify* is a hosting platform with a git-based workflow. Homepage: <https://www.netlify.com/>.

## 5. Implementation

```
1 mkdir fds
2 cd ./fds
3 npm init -y
```

**Listing 5.2:** *Creating a new Node.JS program (Shell)*

To use the TypeScript language rather than plain JavaScript, the TypeScript compiler needs to be installed and used in this project. The TypeScript compiler can also be configured to be more suitable to the personal preferences of the developer as well as the requirements of third party libraries used by the project. To install TypeScript as a development dependency<sup>7</sup> and to initiate the configuration file of the TypeScript compiler, run the following commands in a shell terminal:

```
1 npm i -D typescript
2 tsc --init
```

**Listing 5.3:** *Installing and configuring TypeScript compiler (Shell)*

To install Express.JS in the current project, run the following command in a shell terminal:

```
1 npm i express
```

**Listing 5.4:** *Installing Express.JS (Shell)*

### Database Connection with ORM (Prisma)

To set up a database connection with Prisma (the ORM library of choice for FDS), the Prisma package should be installed in the current project by running the following command in a shell terminal

```
1 npm i -D prisma
2 npm i @prisma/client
```

**Listing 5.5:** *Installing Prisma (Shell)*

After installing the Prisma package, set up the current project to work with Prisma ORM by running the following command in a shell terminal:

```
1 npx prisma init
```

**Listing 5.6:** *Set up project with Prisma*

After the project is set up, a `prisma.schema` file will be created. The `prisma.schema` file is used as a configuration file for the Prisma client and to define the database schema. The database type as well as its URL should be configured in this `prisma.schema` file.

```
1 datasource db {
```

---

<sup>7</sup>In a Node.JS project, development dependency is a third party library used only for development purposes.

## 5. Implementation

```
2 |     provider = "mongodb"
3 |     url       = env("DB_URL")
4 | }
```

**Listing 5.7:** *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 set by the `DB_URL` environment variable.

As the configuration is done and the database schema is created, the Prisma client API needs to be updated to include the interfaces generated from the schema. To update the Prisma client API, run the following command in a shell terminal:

```
1 | npx prisma generate
```

**Listing 5.8:** *Update Prisma client API to include generated interfaces from the schema (Shell)*

### Tsoa and Swagger

Even though Express.JS provides a declarative and easy way to build a server side application, it's built with JavaScript in mind. Therefore, even though TypeScript can be used with Express.JS, it doesn't use all the potential functionalities that TypeScript provides. *Tsoa*<sup>8</sup> 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<sup>9</sup> specification based on HTTP controller code
- Generate Swagger schema based on TypeScript interfaces
- Generate Swagger schema descriptions based on *jsDoc*<sup>10</sup> comments on the source code

Other than that, *tsoa* provides an alternative syntax to build an Express.JS application in a more object-oriented way. *Tsoa* works by compiling the code, with the help of the TypeScript compiler into a regular Express.JS application, built with JavaScript.

### 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<sup>11</sup>.

### Docker

The application will be built and run as a Docker container. To be able to run an application as a Docker container, a `Dockerfile` is needed to provide a list of commands

<sup>8</sup>*Tsoa* GitHub repository: <https://github.com/lukeautry/tsoa>.

<sup>9</sup>*Swagger* is a tool to document server side APIs. Homepage: <https://swagger.io/>.

<sup>10</sup>*jsDoc* is a tool to generate API documentation, similar to *JavaDoc*. Homepage: <https://jsdoc.app/>.

<sup>11</sup>In the UI project, there are certain code style rules regarding HTML elements.

## 5. Implementation

needed to be run to assemble the particular image. The commands used to assemble a Docker container for the FDS are:

```
1 FROM node:16-alpine
2
3 ARG ARG_1 # Arguments passed by --build-args flag
4 ENV ARG_1=$ARG_1 # Environment variable of the image
5
6 WORKDIR /app
7 COPY ["package.json", "package-lock.json*", "./"]
8
9 RUN npm ci # Install packages
10 COPY . .
11 # Additional steps to setup the application
12
13 RUN npm run build
14 ENV NODE_ENV=production
15
16 CMD [ "node", "./dist/src/main.js" ]
```

**Listing 5.9:** 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*<sup>12</sup>. 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:

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

**Listing 5.10:** Running a RabbitMQ instance with Docker (Shell)

---

<sup>12</sup>*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>.

## 5. Implementation

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*<sup>13</sup> 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:

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

**Listing 5.11:** *Running a Redis instance with Docker (Shell)*

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

### 5.1.5. MongoDB

MongoDB is the database of choice for the system. MongoDB has an official Docker image and the following command should be run in a shell terminal to set up a MongoDB instance locally:

```
1 docker run --name mongoddb -d -p 27017:27017 mongo
```

**Listing 5.12:** *Running a MongoDB instance with Docker (Shell)*

By running the command listed above, a MongoDB instance will be run locally on port 27017. On the live environment, a service called *MongoDB Atlas*<sup>14</sup> will be 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

**TODO. Implementation not done**

## 5.2. Model

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

---

<sup>13</sup>*CloudAMQP* is a service provider (RabbitMQ as a Service) that offers RabbitMQ clusters setup and management.

<sup>14</sup>*MongoDB Atlas* is a service that provides a cloud-hosted MongoDB instances. Homepage: <https://www.mongodb.com/atlas>.

## 5. Implementation

### 5.2.1. Validation Rule

To implement the model defined in Figure 4.8, a TypeScript interface is created to help in providing a clear structure during development phase. A *Prisma* schema<sup>15</sup> is also created to sync the structure of the data saved in database with the latest TypeScript interface.

```
1 // fds/src/types/rule.d.ts
2
3 export interface ValidationRule {
4   retryStrategy?: RetryStrategy | null
5   requestUrlParameter?: GenericObject
6   requestHeader?: GenericObject
7   skip: boolean
8   requestBody?: GenericObject
9   condition: Condition | BooleanCondition
10  method: "GET" | "PUT" | "POST"
11  failScore: number
12  endpoint: string
13  priority: number
14  name: string
15 }
16
17 type GenericObject = { [key: string]: any } // Dictionary
18 type Condition = {
19   path: string
20   operator: OperatorType
21   type: ConditionType
22   value: any
23   failMessage: string
24 }
25
26 type BooleanCondition = {
27   all: Condition[]
28 } | {
29   any: Condition[]
30 }
31
32 type RetryStrategy = {
33   limit: number
34   statusCodes: number[]
35 }
36
```

**Listing 5.13:** TypeScript interface of a validation rule (TypeScript)

### 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*<sup>16</sup> to make HTTP requests

<sup>15</sup>The *Prisma* database schema is listed on Listing A.1

<sup>16</sup>*Got* is a Node.js library to make HTTP requests. GitHub repository: <https://github.com/sindresorhus/got>.

## 5. Implementation

to the external endpoints. Got provides the functionality to retry a failed HTTP request out of the box<sup>17</sup>.

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.

### Validation Rules Management Using ORM (Prisma)

The management of validation rules entries in the database are handled by the ORM library of choice (Prisma). This includes creating, reading, updating and deleting the entries. The schema of a validation rule 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`.

```
1 import { PrismaClient } from '@prisma/client'
2
3 const main = async () => {
4   const prisma = new PrismaClient() // Create new prisma instance
5   await prisma.$connect() // Establish connection to database
6 }
7
```

**Listing 5.14:** *Establishing database connection with Prisma (TypeScript)*

The ORM simplifies the access to the database entries by providing type-safe interfaces and provides a layer of abstraction on top of the database system.

```
1 await prisma.validationRule.findMany()
2 await prisma.validationRule.delete({
3   where: {
4     name: validationRule.name
5   }
6 })
7
```

**Listing 5.15:** *Example usage of Prisma (TypeScript)*

### Model Validation

TODO: Add model validation. <https://github.com/BA-LouisAndrew/fds/issues/70>

#### 5.2.2. Validation Result

A TypeScript interface is created as the implementation of 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.

---

<sup>17</sup>For more information on Got's retry options, please refer to <https://github.com/sindresorhus/got/blob/main/documentation/7-retry.md>.



## 5. Implementation

```
1  export interface Validation<T> {
2      validationId: string
3      fraudScore: number
4      totalChecks: number
5      runnedChecks: number
6      skippedChecks: string[]
7      additionalInfo: ValidationAdditionalInfo<T>
8      events: ValidationEvent[]
9  }
10
11  export type ValidationEventStatus = "NOT_STARTED" | "FAILED" | "PASSED" | "RUNNING"
12  export type ValidationEvent = {
13      name: string
14      status: ValidationEventStatus
15      dateStarted: string | null
16      dateEnded: string | null
17      messages?: string[]
18  }
19
20  export type ValidationAdditionalInfo<T> = {
21      startDate: string
22      endDate?: string
23      customerInformation?: T
24  }
25
```

**Listing 5.16:** *TypeScript interface of a validation result (TypeScript)*

### 5.3. Controller

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. Customer Validation on a Registration Event

An HTTP endpoint will be 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.

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 subsection 5.3.2). 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. Implementation

### 5.3.2. 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<sup>18</sup> 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 `scheduleRulesetValidation` method. Before running a validation process, the list of validation rules as well as runtime secrets<sup>19</sup> should be provided to the engine. The `ValidationEngine` class uses method chaining<sup>20</sup> as well as the *Builder* design pattern discussed in [4, pp. 97-106] for its construction, to make the public API of a validation engine as simple as possible.

```
1 export class ValidationEngine<T> {
2   private secrets: GenericObject = {}
3   private ruleset: ValidationRule[] = []
4
5   setSecrets(secrets: GenericObject) {
6     this.secrets = secrets
7     return this
8   }
9
10  setRuleset(ruleset: ValidationRule[]) {
11    this.ruleset = [...ruleset.sort((a, b) => b.priority - a.priority)]
12    return this
13  }
14 }
15
```

**Listing 5.17:** *ValidationEngine* class builder pattern using method chaining (TypeScript)

<sup>18</sup>Not skipped.

<sup>19</sup>Runtime secrets are used to store confidential information such as API keys.

<sup>20</sup>*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.

## 5. Implementation

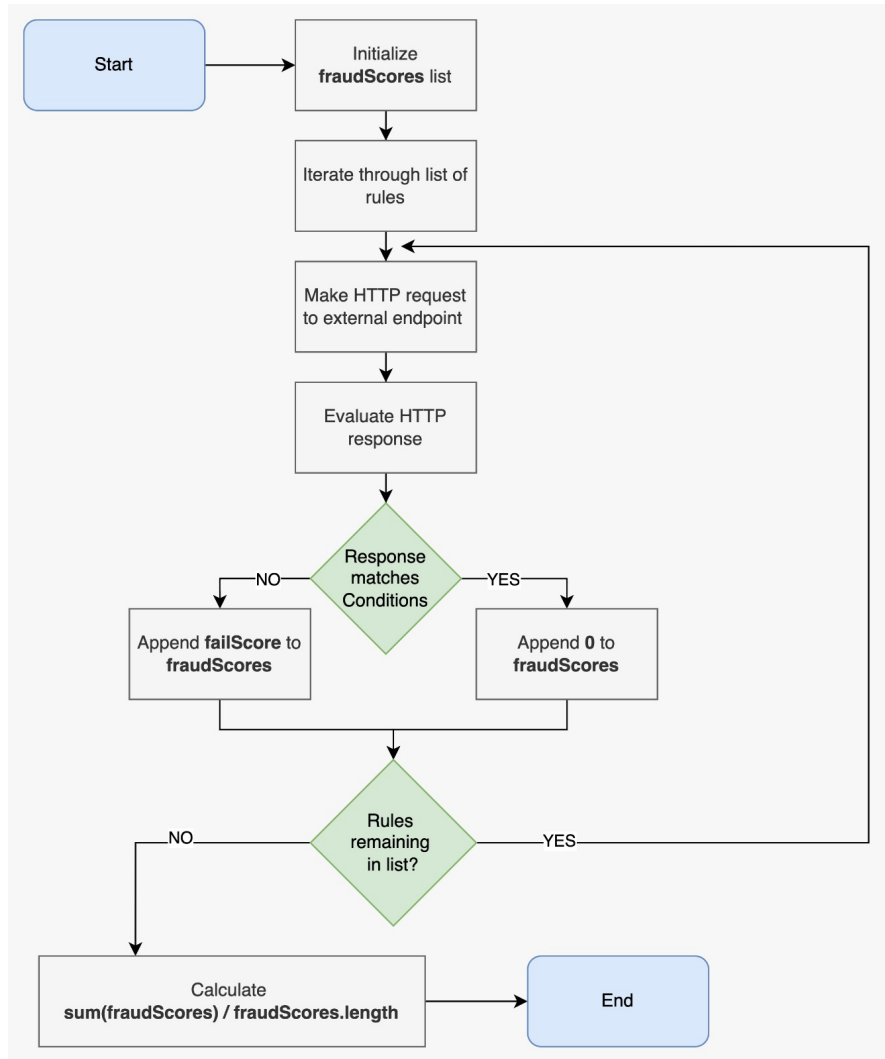


Figure 5.1.: Flow diagram of a validation process

### 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 the external endpoint of the particular rule.

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

```
1 import jp from "jsonpath"
2
3 const accessDataFromPath = (runtimeData: any, expression: any) => {
4   if (typeof expression !== "string") {
5     return expression // A valid JSONPath expression is a string
```

## 5. Implementation

```
6   }
7
8   try {
9     const [dataFromPath] = jp.query(runtimeData, expression)
10    if (!dataFromPath) {
11      // Expression is valid, but the path doesn't point to a specific value
12      return expression
13    }
14
15    return dataFromPath
16  } catch {
17    return expression // The expression is either not a valid JSONPath expression
18  }
19 }
20
```

**Listing 5.18:** 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 [4, pp. 127-134], as there might 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 a Got instance, used to make HTTP requests. In testing environment, the context object contains a mocked Got instance.

```
1 export class Agent {
2   private static context: Context // Dependency injection
3
4   private static get client() {
5     return Agent.context.client // `client` object is a Got instance
6   }
7
8   static setClient(context: Context) {
9     this.context = context
10  }
11 }
12
```

**Listing 5.19:** Usage of the singleton pattern and dependency injection in Agent class (TypeScript)

## 5. Implementation

### 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 on its group of type. 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 operate function 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.

```
1 export class NumberOperator extends Operator<number, NumberOperatorIds> {  
2   const validateFunction = (value) =>  
3     typeof value === "number" &&  
4       !isNaN(parseFloat(`${value}`))  
5 }  
6  
7 export const numberOperators: Record<NumberOperatorIds, NumberOperator> = {  
8   eq: new NumberOperator("eq", (a, b) => b === a),  
9   gt: new NumberOperator("gt", (a, b) => b > a),  
10  gte: new NumberOperator("gte", (a, b) => b >= a),  
11  lt: new NumberOperator("lt", (a, b) => b < a),  
12  lte: new NumberOperator("lte", (a, b) => b <= a),  
13 }  
14
```

Listing 5.20: NumberOperator example (TypeScript)

The *Flyweight* design pattern mentioned in [4, 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 regarding a certain condition. An evaluator works together with the Operator class in evaluating the conditions given. The specific operator to evaluate a condition is accessed via the OperatorFactory. The Evaluator class encapsulates the internal logic of evaluating a condition and running the operations defined by the particular condition. The Evaluator class is also responsible in

## 5. Implementation

accessing the required data described by a JSONPath expression from the runtime information of a validation process.

```
1 // Evaluate JSONPath expressions.
2 const dataFromPath = this.accessDataFromPath(runtimeData, validationRule.path)
3 const valueFromPath = this.accessDataFromPath(runtimeData, validationRule.value)
4
5 const operator = OperatorFactory.getOperator(
6   validationRule.type,
7   validationRule.operator,
8 )
9 const isEvaluationPassed = operator.operate(valueFromPath, dataFromPath)
10
```

**Listing 5.21:** *The usage of OperatorFactory class in the Evaluator class (TypeScript)*

As mentioned before, 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 is responsible in evaluating a single condition, while the BooleanConditionEvaluator is in charge 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 [4, pp. 107-116].

```
1 const response = await Agent.fireRequest(rule, {
2   customer: customerData,
3   secrets: this.secrets
4 })
5
6 const evaluator = EvaluatorFactory.getEvaluator(condition)
7 const evaluationResult = evaluator.evaluate({
8   response: response.data,
9   customer: customerData,
10  secrets: this.secrets
11 })
12
```

**Listing 5.22:** *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 fails).

### 5.3.3. Notification on Suspicious Cases

In [16], Subramoni et al. describes an exchange as a routing mediator that copies and send the message sent 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

## 5. Implementation

access and publish a message to the exchange is implemented using the *Singleton* [4, 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 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.

```
1 import { Channel, connect } from "amqplib"
2 export class Notification {
3   private static channel: Channel | null = null
4
5   async init(url: string) {
6     try {
7       const connection = await connect(url)
8       const channel = await connection.createChannel() // Create a new channel
9       // Assert whether the exchange exists, create new if it doesn't exist
10      await channel.assertExchange("FDS", "fanout", {
11        durable: true
12      })
13
14      Notification.channel = channel
15    } catch (err) {
16      console.error(err)
17    }
18  }
19 }
20
```

**Listing 5.23:** *Opening 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.

```
1 Notification.publish(JSON.stringify(this.validationResult))
2
```

**Listing 5.24:** *Publishing a validation result to the RabbitMQ exchange (TypeScript)*

There can be many consumers consuming the messages published to the exchange, and run certain actions regarding the internal logic of the system itself. 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 needs to be opened, and a dedicated message queue (ideally created only for internal usage of the consumer itself) is needed.

## 5. Implementation

```
1 import { connect } from "amqplib"
2 export const start = async (url: string) => {
3   try {
4     const connection = await connect(url)
5     const channel = await connection.createChannel()
6     await channel.assertExchange("FDS", "fanout", { durable: true })
7     // Create an exclusive queue (created only for internal usage)
8     const { queue } = await channel.assertQueue("", { exclusive: true })
9
10    // Bind the exclusive queue to the exchange
11    channel.bindQueue(queue, "FDS", "")
12
13    await channel.consume(queue, async (message: string) => {
14      // Do actions
15    }, {
16      noAck: true
17    })
18  } catch (err) {
19    console.error(err)
20  }
21 }
22 }
```

**Listing 5.25:** Consuming a message published to the RabbitMQ exchange (TypeScript)

### 5.3.4. 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* [4, 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<sup>21</sup>.

### 5.3.5. Validation Real Time Progress

The *Observer* [4, pp. 293-303] will also be implemented here to provide the functionality of a real time progress update of a certain validation process. An *EventEmitter* is used here to subscribe 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 on the name of the event 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 the name of the event, as an indicator that the validation process is completed and no further events with the attached validation ID will be sent.

---

<sup>21</sup>Runtime secrets are not cached.



## 5. Implementation

```
1 // Published when a rule evaluation is completed
2 EventBus.emit(
3   `${EventBus.EVENTS.VALIDATION_EVENT_UPDATE}--${validationId}`,
4   this.validationResult,
5 )
6 // Published when a validation process is completed
7 EventBus.emit(`${EventBus.EVENTS.VALIDATION_DONE}--${this.validation.validationId}`)
8
```

**Listing 5.26:** *Publishing events to the EventEmitter on certain validation events (TypeScript)*

To continually provide a client with the latest progress of a validation event without having to provide a dedicated instance of a WebSocket server, the Server-Sent Events (SSE)[14] 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. SSE is enough for the use case of the project, as the client doesn't need to send new data to the client to display real time progress of a validation process.

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 message 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.

```
1 static async subscribeToValidationProgress (
2   validationId: string,
3   responseObject: Response,
4 ) {
5   const updateEvent = `${EventBus.EVENTS.VALIDATION_EVENT_UPDATE}--${validationId}`
6   const closeEvent = `${EventBus.EVENTS.VALIDATION_DONE}--${validationId}`
7
8   EventBus.once(closeEvent, () => {
9     closeConnection()
10  })
11
12  EventBus.on(
13    updateEvent,
14    (validationResult: Validation) => {
15      writeToStream(validationResult)
16    },
17  )
18
19  const writeToStream = (data: any) =>
20    responseObject.write(
21      `data: ${JSON.stringify(data)}\n\n`,
22    )
23 }
```

## 5. Implementation

```
23 | }  
24 |
```

**Listing 5.27:** *Writing to SSE stream when certain events are published (TypeScript)*

### 5.4. View

Optionals:

- Architecture -> (can be in controller)
- Techs

## 6. Test

[Beschreibung, wie Sie auf Basis des geplanten Testvorgehens was mit welchen Kriterien und Technologien getestet haben]

## 7. Demonstration and Evaluation

[Beschreibung der Ergebnisse aus allen voran gegangenen Kapiteln sowie der zuvor generierten Ergebnisartefakte mit Bewertung, wie diese einzuordnen sind]

### **7.1. Ausblick**

[Beschreibung und Begründung potenzieller zukünftiger Folgeaktivitäten im Zusammenhang mit Ihrer Arbeit (z.B. weitere Anforderungen, Theoriebildung, ... )]

# Bibliography

- [1] Helmut Balzert, Marion Schröder, and Christian Schaefer. *Wissenschaftliches Arbeiten. Ethik, Inhalt & Form wiss. Arbeiten, Handwerkszeug, Quellen, Projektmanagement, Präsentation*. 2. Auflage. Herdecke, Witten: W3L, 2011. ISBN: 978-3-86834-034-1.
- [2] Norbert Franck and Joachim Stary. *Die Technik wissenschaftlichen Arbeitens: eine praktische Anleitung*. 17. Auflage. Paderborn: Schöningh, 2013. ISBN: 978-3-50697-027-5.
- [3] 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](https://doi.org/10.1007/978-1-4842-4330-5_10).
- [4] Erich Gamma et al. *Design Patterns*. Addison-Wesley, 1995.
- [5] David Garlan. “Software architecture”. In: *Proceedings of the conference on The future of Software engineering - ICSE '00* (2000). DOI: 10.1145/336512.336537.
- [6] ISO. *ISO/IEC 25010:2011*. Mar. 2011. URL: <https://www.iso.org/standard/35733.html>.
- [7] Glenn Krasner and Stephen Pope. “A cookbook for using the model-view controller user interface paradigm in Smalltalk-80”. In: *Journal of Object-oriented Programming - JOOP* 1 (Jan. 1988).
- [8] Lorient. *Möpsen und Menschen. Eine Art Biographie. Zürich*. In: faz.net. Online: [https://media0.faz.net/ppmedia/aktuell/feuilleton/1461387463/1.721778/format\\_top1\\_breit/die-steinlaus-trotzt-seit.jpg](https://media0.faz.net/ppmedia/aktuell/feuilleton/1461387463/1.721778/format_top1_breit/die-steinlaus-trotzt-seit.jpg); letzter Zugriff: 13 VI 19. 1983.
- [9] Lorient. *Steinlaus, Lorient Katalog, Diogenes Verlag, Zürich*. In: tagblatt.de. Online: <https://www.tagblatt.de/Bilder/Lorients-legendaere-Steinlaus-Lorient-Katalog-1993-2003-125217h.jpg>; letzter Zugriff: 14 VI 19. 1993, 2003.
- [10] Alexey Melnikov and Ian Fette. *The WebSocket Protocol*. RFC 6455. <http://www.rfc-editor.org/rfc/rfc6455.txt>. RFC Editor, 2011. URL: <http://www.rfc-editor.org/rfc/rfc6455.txt>.
- [11] Henrik Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. Tech. rep. 2616. June 1999. 176 pp. DOI: 10.17487/RFC2616. URL: <https://www.rfc-editor.org/info/rfc2616>.
- [12] 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](https://doi.org/10.1007/978-1-4302-6269-5_15).

## Bibliography

- [13] Pschyrembel online. *Steinlaus*. Online: <https://www.pschyrembel.de/Steinlaus/KOLHT>; letzter Zugriff: 14 VI 19. 2016.
- [14] Wendy Roome and Y. Richard Yang. *Application-Layer Traffic Optimization (ALTO) Incremental Updates Using Server-Sent Events (SSE)*. RFC 8895. Nov. 2020. DOI: 10.17487/RFC8895. URL: <https://www.rfc-editor.org/info/rfc8895>.
- [15] Alan Jay Smith. "Cache Memories". In: *ACM Computing Surveys* 14.3 (1982), pp. 473–530. DOI: 10.1145/356887.356892.
- [16] 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.
- [17] Wikipedia. *Academic Use*. Online: [https://en.wikipedia.org/wiki/Wikipedia:Academic\\_use](https://en.wikipedia.org/wiki/Wikipedia:Academic_use); letzter Zugriff: 13 VI 19. 2019.

## 8. List of Abbreviations



## 9. Glossary

# A. Appendix

## A.1. Supplemental Figures

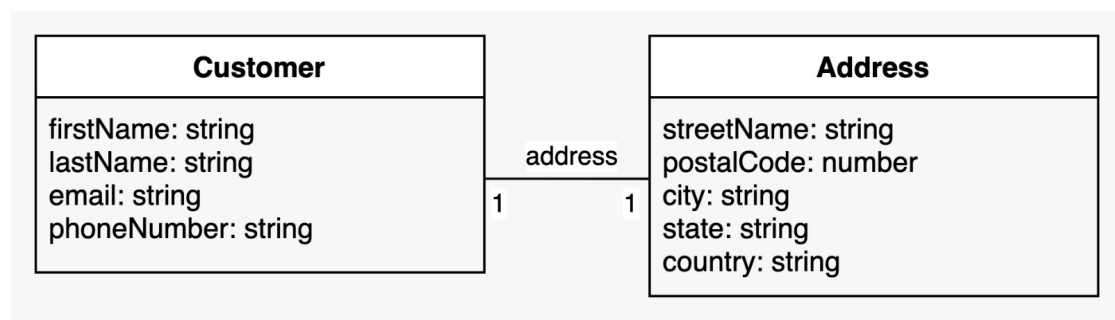


Figure A.1.: UML diagram of the customer model

## A.2. Supplemental Source Codes

```
1  model ValidationRule {
2    id          String      @id @default(auto()) @map("_id") @db.
      ObjectId
3    name        String      @unique
4    skip        Boolean
5    priority    Int
6    endpoint    String
7    method      String
8    failScore   Float
9    condition   Json
10   retryStrategy Json?
11   requestUrlParameter Json?
12   requestBody  Json?
13   requestHeader Json?
14 }
```

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

## A.3. Quell-Code

## A.4. Tipps zum Schreiben Ihrer Abschlussarbeit

- Achten Sie auf eine neutrale, fachliche Sprache. Keine „Ich“-Form.

## A. Appendix

- Zitieren Sie zitierfähige und -würdige Quellen (z.B. wissenschaftliche Artikel und Fachbücher; nach Möglichkeit keine Blogs und keinesfalls Wikipedia<sup>1</sup>).
- Zitieren Sie korrekt und homogen.
- Verwenden Sie keine Fußnoten für die Literaturangaben.
- Recherchieren Sie ausführlich den Stand der Wissenschaft und Technik.
- Achten Sie auf die Qualität der Ausarbeitung (z.B. auf Rechtschreibung).
- Informieren Sie sich ggf. vorab darüber, wie man wissenschaftlich arbeitet bzw. schreibt:
  - Mittels Fachliteratur<sup>2</sup>, oder
  - Beim Lernzentrum<sup>3</sup>.
- Nutzen Sie L<sup>A</sup>T<sub>E</sub>X<sup>4</sup>.

---

<sup>1</sup>Wikipedia selbst empfiehlt, von der Zitation von Wikipedia-Inhalten im akademischen Umfeld Abstand zu nehmen [17].

<sup>2</sup>Z.B. [1], [2]

<sup>3</sup>Weitere Informationen zum Schreibcoaching finden sich hier: <https://www.htw-berlin.de/studium/lernzentrum/studierende/schreibcoaching/>; letzter Zugriff: 13 VI 19.

<sup>4</sup>Kein Support bei Installation, Nutzung und Anpassung allfälliger L<sup>A</sup>T<sub>E</sub>X-Templates!

## 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