*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 27 11:26:38 UTC 2022

Abstract

[Summary of the thesis]

Contents

List of Figures

List of Tables

Listings

Listings

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[1]. Quellen können manuell referenziert und im Quellenverzeichnis eingetragen werden. Ergänzend bieten viele Textverarbeitungssteme auch ausgelagerte Quellenverwaltungsdateien und - systeme an, über die mittels entsprechender Befehle im Textteil zitiert werden kann[2].

Visualisieren Sie im Textteil angemessen, z.B. mittels Abbildungen und Tabellen. Vorliegendes Template enth"alt beispielhaft eingebundene Abbildungen und eine Tabelle (vgl. f.), welche der Steinlausforschung[3] entnommen sind.

---

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

[2] 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.

[3] Analog zu Straube (In: [12]) handelt es sich bei der Steinlaus (*petrophaga lorioti*) um das »kleinste einheimische Nagetier«. Als stimmungsaufhellender Endoparasit erreicht es eine Grö"se 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.
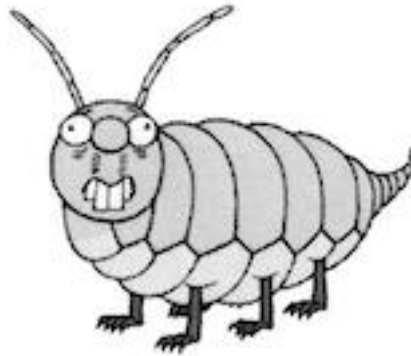
Figure 1.1.: *Example image: Who is Steinlaus?; Bildquelle [8]*
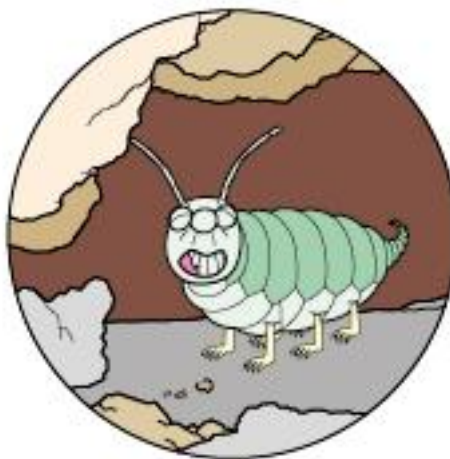
Figure 1.2.: *Beispielgrafik: Fressende Steinlaus; Bildquelle [7]*

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

1.1. 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

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

3.1. Goal

The system should be able to provide a possibility for different teams to contribute on the fraud detection process independently, based on their views and knowledge on the characteristics of a potentially fraudulent customer. In such a way, the system can help in transforming the fraud detection process to be a collaborative process and each team make a contribution by making it more reliable as their knowledge increases. Primarily, the system should enable collaboration across autonomous teams in a fraud detection process and simplify it by encapsulating the internal logic while presenting the result in an understandable format.

3.2. System Environment

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

3.3. State of the Art

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

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

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

3.4. Requirements

4. Conception and Design

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

4.1. System Architecture

A system architecture diagram is created to help to understand the system as well as its components better.
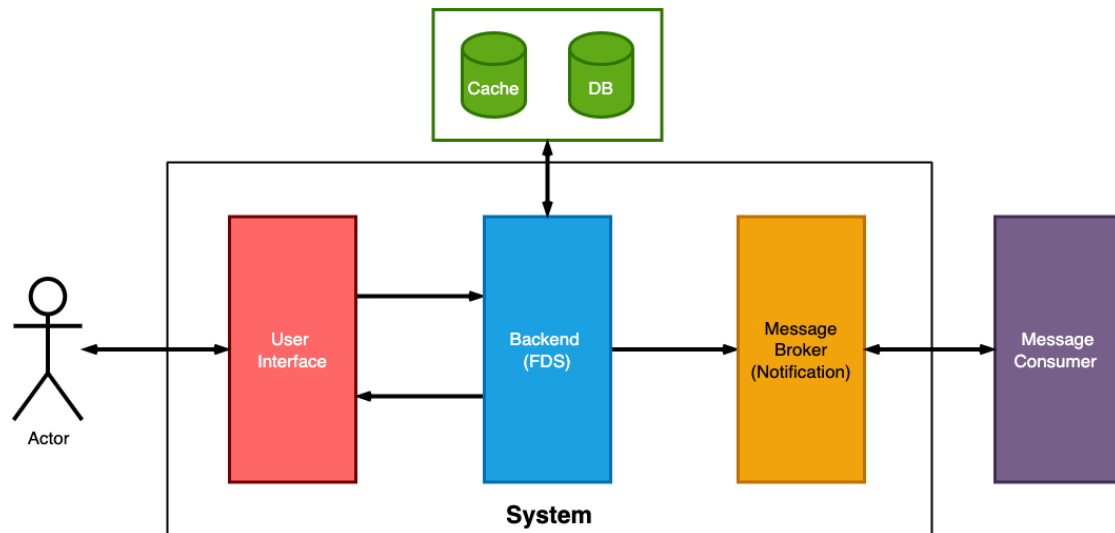
Figure 4.1.: *System architecture diagram*

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

---

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

4.2. Software Architecture

As the requirement is clear and the components of the system are defined, a software architecture is needed. A software architecture plays a major role in a software development project by providing a structure on how the software should be built and decisions made during this stage would be vital for the development process going forward. As Garlan wrote in one of his work *Software Architecture*, a software architecture "plays a key role as a bridge between requirements and implementation."[5] A software architecture diagram is therefore created to help visualize the structure, functions and role of each component of the system.
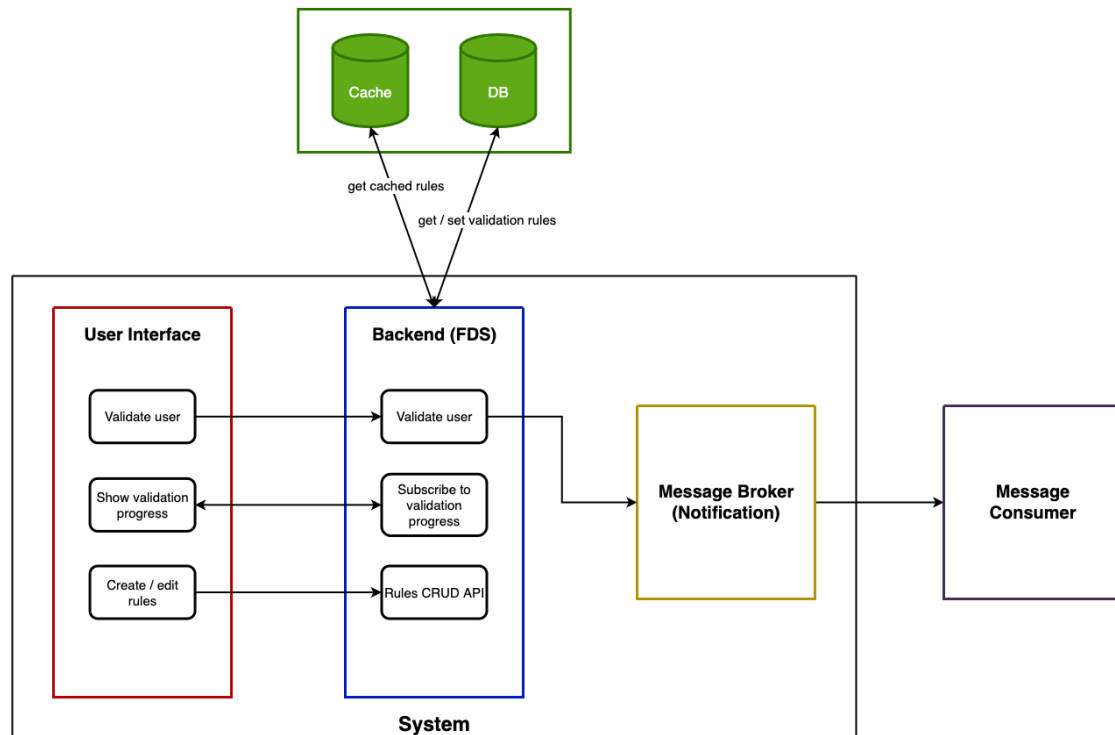
Figure 4.2.: *Software architecture diagram*

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

be discussed in subchapter 4.4.2. A database connection should be established on the FDS to persist the validation rules. An optional connection to a cache memory could also be established to enable a faster access to the data needed.

The core functionality of the FDS is to run validation process of a customer by evaluating a collection of validation rules in relation to a given customer data. The execution of the validation process could be scheduled via a single HTTP request that contains the customer data on its request payload[2]. 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[3] 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

---

[2]A request payload refers to data sent by a client to the server during an HTTP request, usually attached to the request body[10, section "4.3 Message Body"].

[3]In [9], 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.

- Database and caching memory
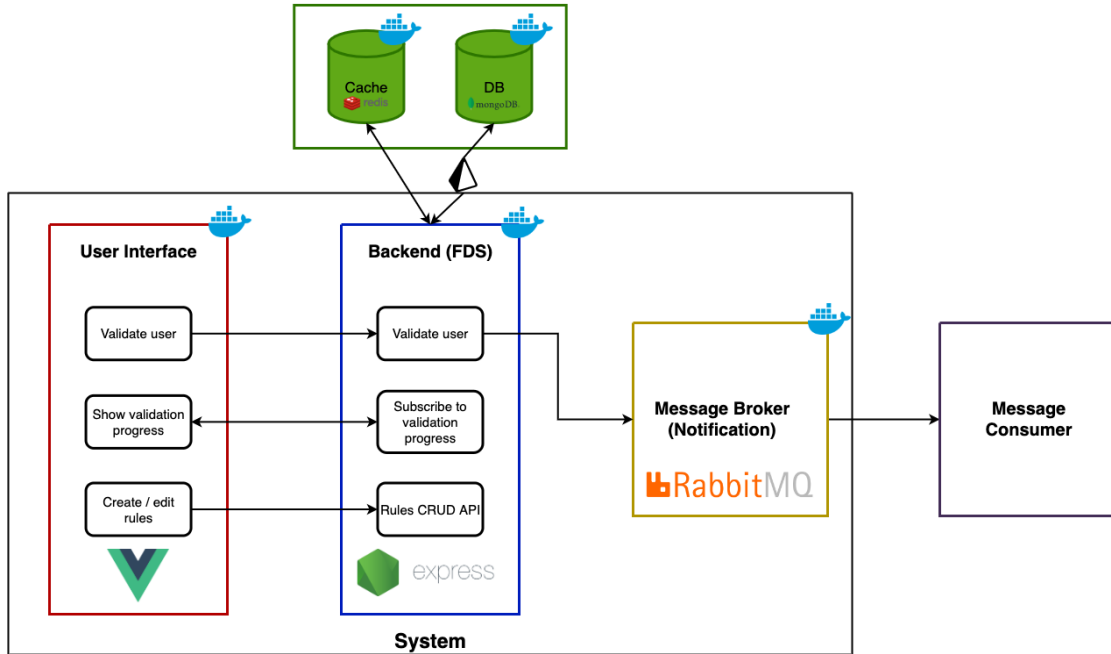
Figure 4.3.: *Software architecture diagram with technologies used*

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[4] 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[5].

4.3.1. User Interface

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

---

[4]*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/.

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

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

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

4.3.2. FDS

The technology chosen to build the FDS is Node.JS[8]. 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[9] 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[10], 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[11], 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[12] will be used as the database system of choice. Redis[13] 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

---

[8]*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`.

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

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

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

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

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

and Pope [6] 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 [6] 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
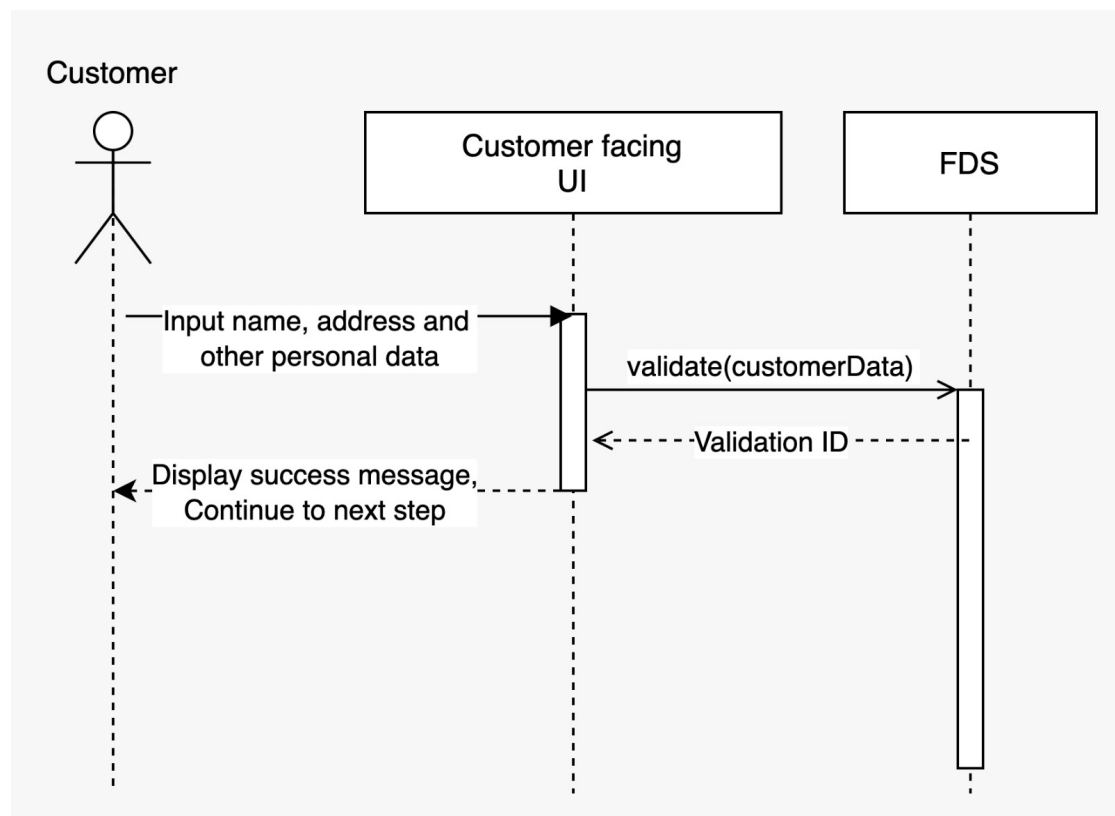
Figure 4.4.: *System sequence diagram for a customer validation when a new customer is registered*

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"

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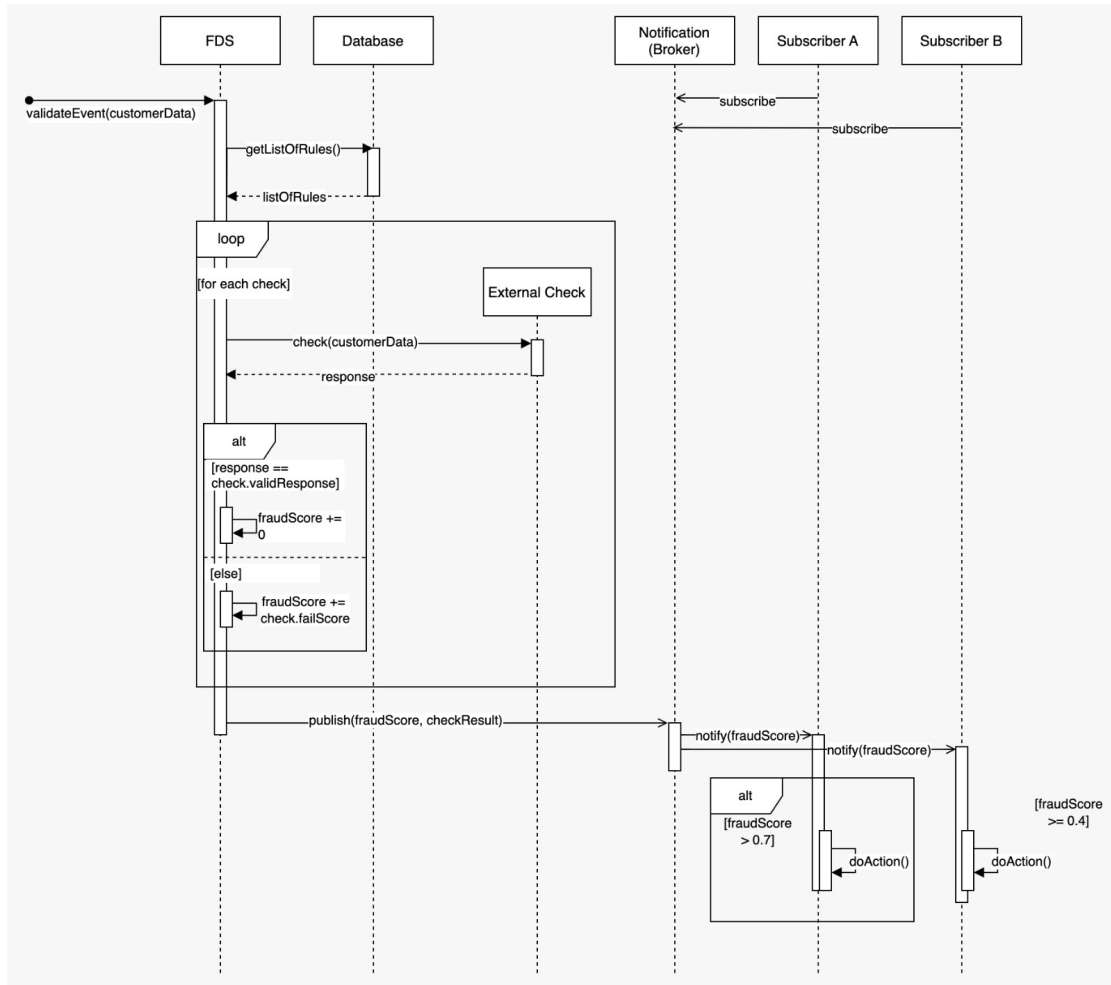
Figure 4.5.: *System sequence diagram for notifications on suspicious cases*

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[14]

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"

---

[14]For example: sending an email notification if the fraud score exceeds 0.7.
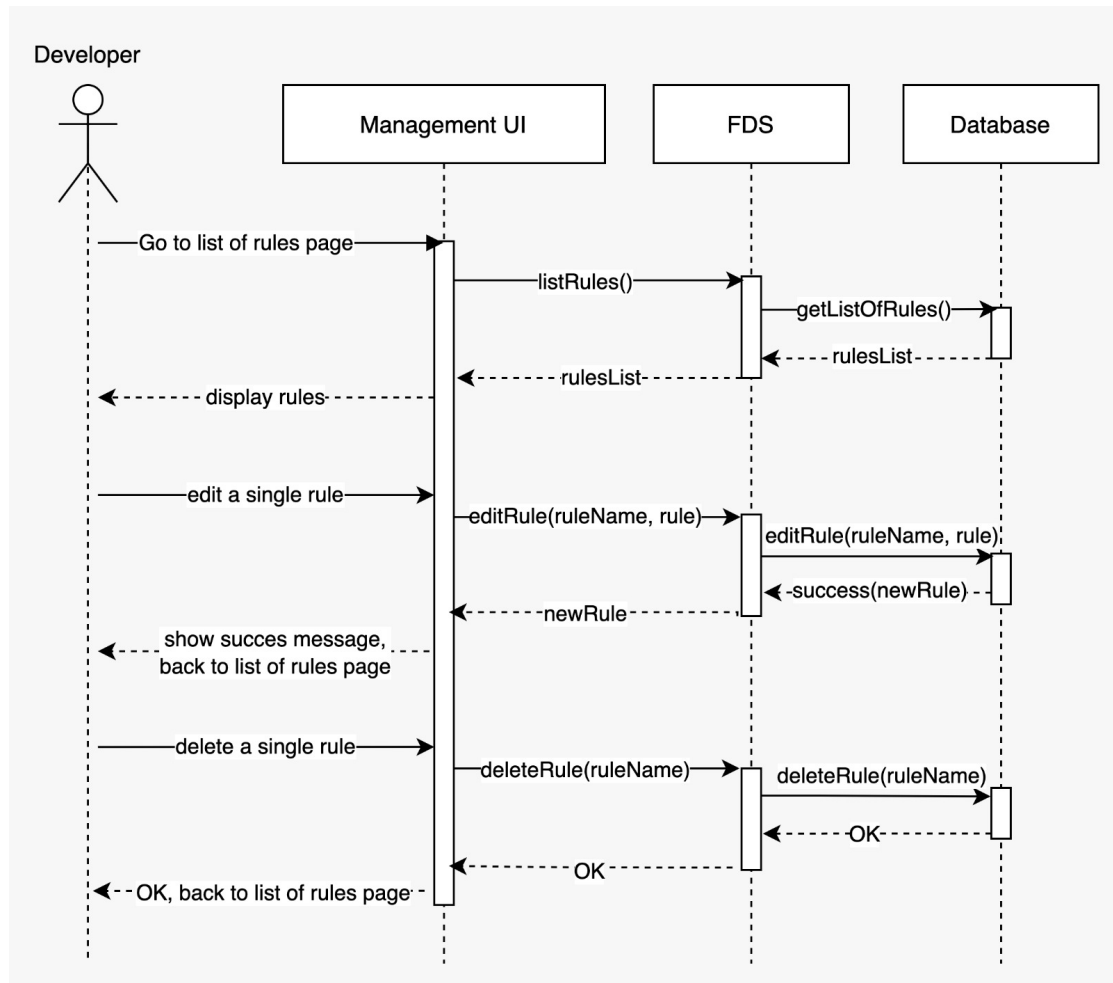
Figure 4.6.: *System sequence diagram for validation rules management*

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[15] 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

---

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

- 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[16] 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[17] 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
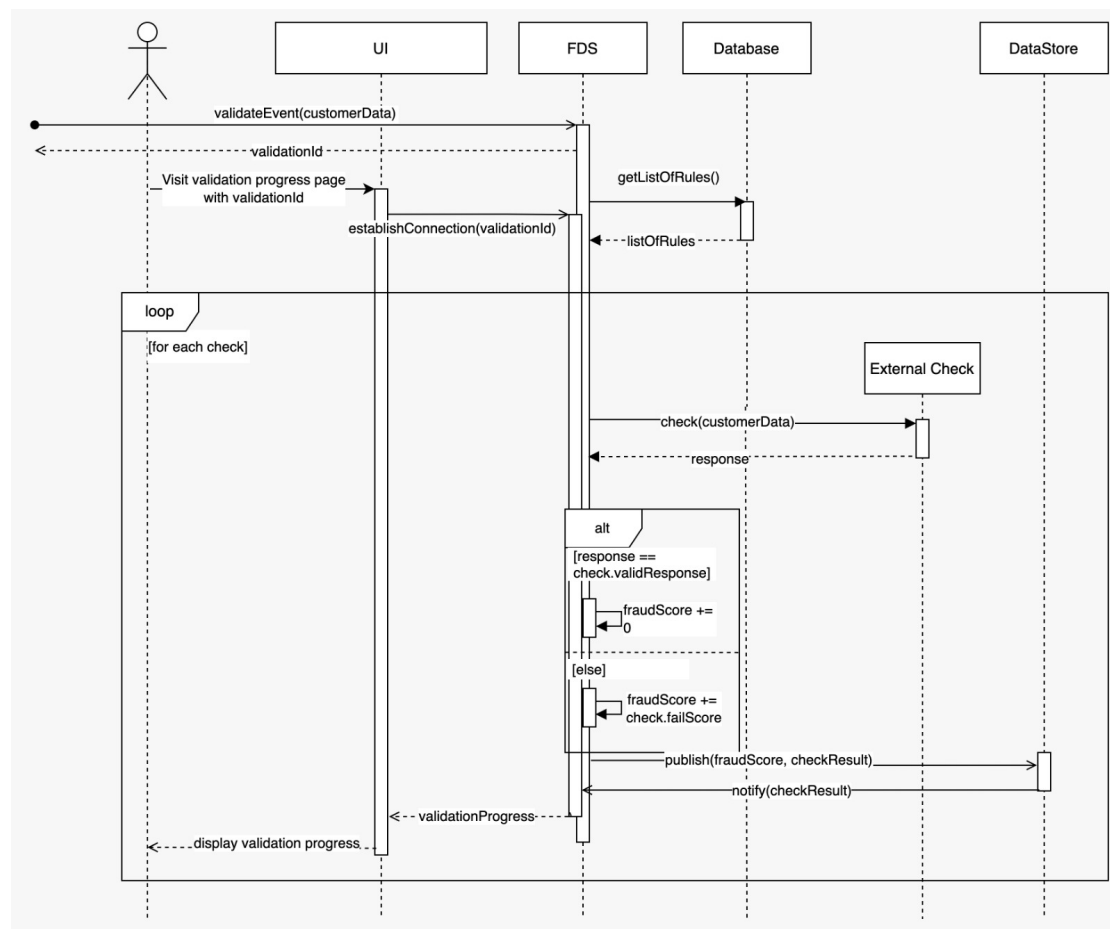
Figure 4.7.: *System sequence diagram for validation rules management*

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.

---

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

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

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[18]
- 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 [6], 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 subchapter 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

---

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

- Request header[19]
- Request body

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

An additional *priority* attribute is also provided to enable the possibility to run rule validations according to its priority order.
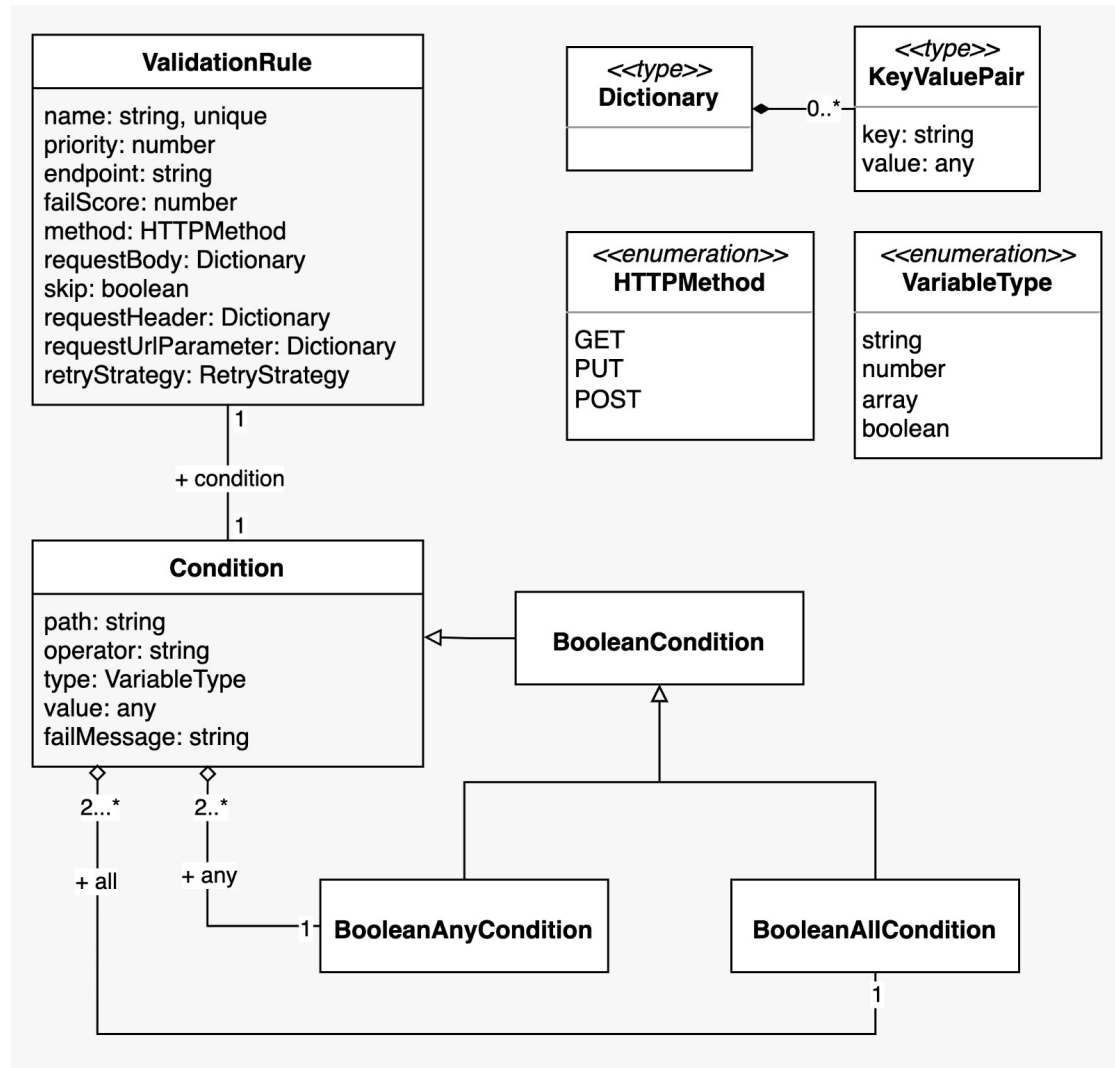
Figure 4.8.: *UML diagram of the validation rule model*

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 evaluation. It is intended to design the condition attribute to be robust and configurable.

---

[19]In [10, "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.

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[20]. The *operator* attribute refers to a name of operator to be used to evaluate the condition (for example: "eq", "incl"). The available operator names are predefined and restricted to the condition's type attribute. More information regarding operators will be discussed in section 5.3.2. 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       {
```

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

```
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 section 5.3.2. By evaluating the sequences listed on Figure 4.4 and Figure 4.5, the following attributes are needed:

- A unique validation ID
- A fraud score

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

Even though the resulting fraud score determines the probability of a customer being a fraud, it is also important to further analyze the actual evaluation result of each validation rule. For example, a certain action can be run if the evaluation of a specific rule failed or the information regarding the rule evaluations can also be used to display the current progress of a validation process (implementation of this specific functionality will be discussed more in subchapter 5.3.5).

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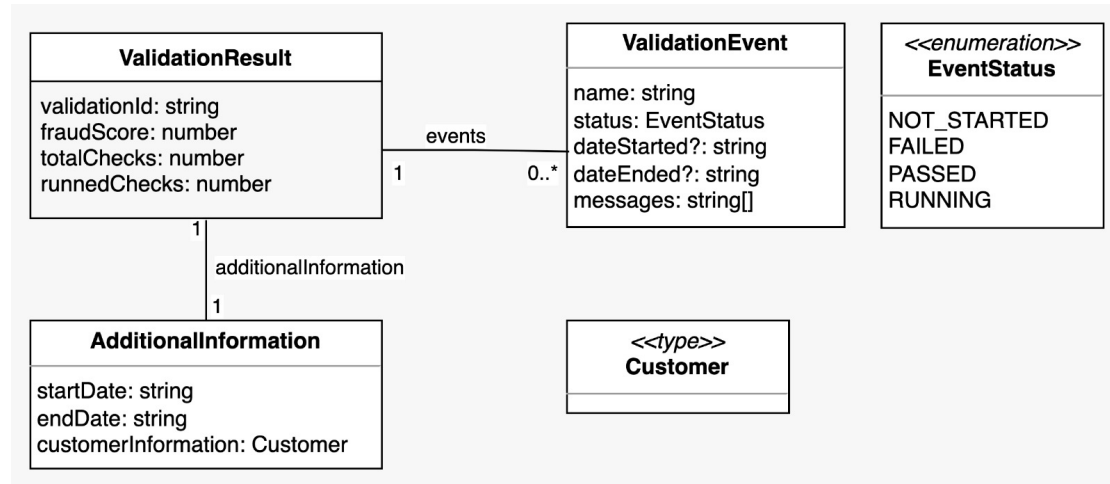
Figure 4.9.: *UML diagram of the validation result model*

```
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",
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 [6], 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[21] to manage a validation rule that will be used to validate a customer.

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

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

---

[21] An understanding on how HTTP works is a prerequisite to use the form.

The CONDITIONS section can be used to add one or more condition to a validation rule. The form fields of each condition includes a dedicated input field for each attribute of a condition, as described in section 4.4.2. The TYPE and OPERATOR form fields are a selectable field, meaning the user has to select one out of several choices provided. This is intended to restrict input from the user, preventing an invalid condition being submitted to the FDS[22]. 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.
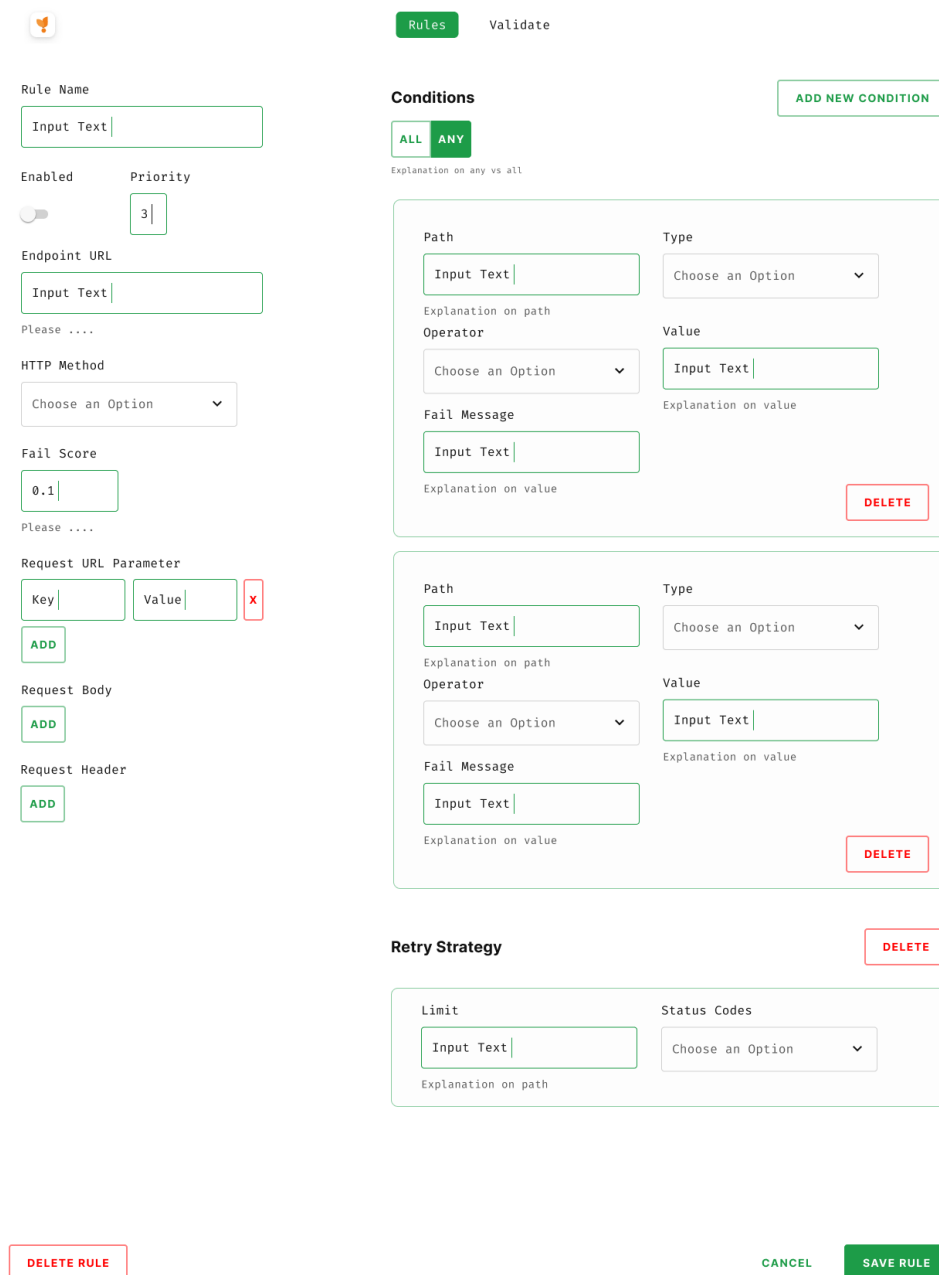
Validation Form

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.

---

[22]For example, on *type* field, user can only choose on of the following: (number, string, array, boolean).

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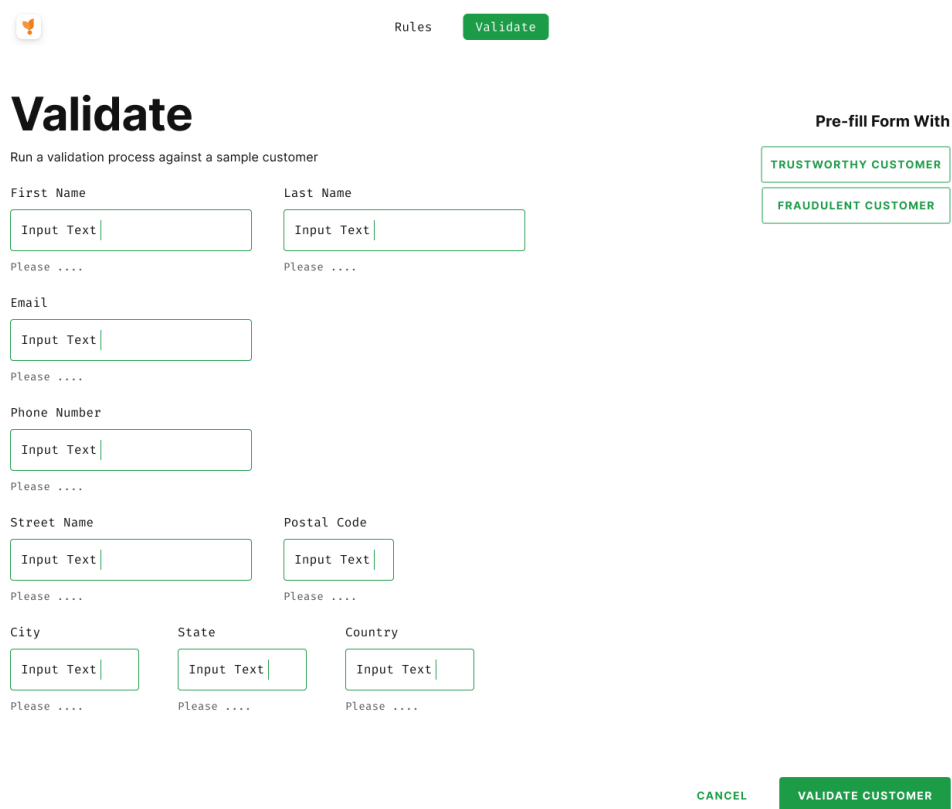
Validation Progress

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

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

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

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

5.1. Technologies and Architecture

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

5.1.1. Prerequisites

There are certain prerequisites before setting up the projects locally. The prerequisites that need to be met are:

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

5.1.2. User Interface

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

Setup

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

---

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

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).

Component Library

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

Code Style

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

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 as a CI/CD platform is *GitHub actions*[4]. The configured CI/CD actions in this project are:

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

Deployment

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

5.1.3. FDS

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

---

[3]*NaiveUI* is a Vue3 component library. GitHub repository: `https://github.com/TuSimple/naive-ui`.
[4]*GitHub actions* is a CI/CD platform, created by GitHub and can be used in all repositories hosted on GitHub. Homepage: `https://github.com/features/actions`.
[5]*Pull request (PR)* is a request from a developer to merge certain changes on a dedicated branch to the main branch of a repository.
[6]*Netlify* is a hosting platform with a git-based workflow. Homepage: `https://www.netlify.com/`.

Setup

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

```
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[7] and to generate a configuration file for 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

---

[7]In a Node.JS project, development dependency is a third party library used only for development purposes.

schema. The type of the database system as well as its URL should be configured in this `prisma.schema` file.

```
1   datasource db {
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 has to be updated to include the interfaces generated from the schema by running `npx prisma generate`.

Tsoa and Swagger

Even though Express.JS provides a declarative and easy way to build a server side application, it's built for a JavaScript application. Although TypeScript can be used with Express.JS, it doesn't use all the potential functionalities that TypeScript provides. *Tsoa*[8] 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[9] specification based on HTTP controller code
- Generate Swagger schema based on TypeScript interfaces
- Generate Swagger schema descriptions based on *jsDoc*[10] comments on the source code
- Validates an HTTP request body based on TypeScript interfaces

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

Code Style

Identical to the UI project, a code formatter (*Prettier*) and linter (*ESLint*) are also used in this project. The configuration for the code formatter and linter is slightly different in comparison to the UI project, as certain code style rules doesn't apply to a server side application[11].

---

[8]*Tsoa* GitHub repository: `https://github.com/lukeautry/tsoa`.

[9]*Swagger* is a tool to document server side APIs. Homepage: `https://swagger.io/`.

[10]*jsDoc* is a tool to generate API documentation, similar to *JavaDoc*. Homepage: `https://jsdoc.app/`.

[11]In the UI project, there are certain code style rules regarding HTML elements.

Docker

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

```
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 5.8: *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*[12]. The deployment process will be executed via the CI/CD action on every commit to the `main` branch.

5.1.4. 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 5.9: *Running a RabbitMQ instance with Docker (Shell)*

---

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

The command listed above will locally run the RabbitMQ instance on port 5672 and the RabbitMQ management UI on port 15672. On the live environment, a service called *CloudAMQP*[13] 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.

5.1.5. 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 5.10: *Running a Redis instance with Docker (Shell)*

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

5.1.6. 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 mongodb -d -p 27017:27017 mongo
```

Listing 5.11: *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*[14] will be used to help host a MongoDB instance in the cloud, making it more accessible by other components of the system.

5.2. Model

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

5.2.1. Validation Rule

The subsection describes the implementation of the `ValidationRule` model as a Type-Script interface and its usage with the ORM library used in the FDS project.

---

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

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

TypeScript interface

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

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

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

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

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

Listing 5.12: *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*[15] to make HTTP requests to the external endpoints. Got provides the functionality to retry a failed HTTP request out of the box[16].

---

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

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

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 validation rule entries in the database. A Prisma schema[17] is created to specify the shape of the data saved in the database. The schema must be created beforehand to generate the required types to be used by the client. After the schema is created, the Prisma Client API should be updated by running `npx prisma generate`.

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

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

Listing 5.13: *Establishing database connection with Prisma (TypeScript)*

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

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

Listing 5.14: *Example usage of Prisma (TypeScript)*

5.2.2. Validation Result

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

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

---

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

```
 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 5.15: *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 subchapter 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.

5.3.2. Validation Process

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

Validation Process Flow

A validation process is started by iterating through a list of validation rules, making an HTTP request to the external endpoint listed on each rule and evaluating its response in comparison to the conditions attached on the rule. If the HTTP response from the

external matches the conditions of the rule, the rule evaluation will be considered as a passed evaluation, otherwise it is a failed evaluation. The result of each rule evaluation determines the value of the resulting fraud score. The resulting fraud score will be calculated as follows:

- Initialize an empty list of fraud scores. The list will be filled later with float numbers ranging between 0 and 1
- Go through the list of validation rules and run evaluation
- If the evaluation passed, append `0` to the list of fraud scores
- Otherwise, append the validation rule `failScore` attribute's value to the list of fraud scores
- At the end of the iteration, the list size should equal to the amount of available[18] validation rules
- The resulting fraud score is the sum of the scores in the list, divided by the number of available validation rules

The flow listed above will be executed by creating a `ValidationEngine` instance and calling a public `secheduleRulesetValidation` method[19]. Before running a validation process, the list of validation rules as well as runtime secrets[20] should be provided to the engine. The `ValidationEngine` class uses method chaining[21] 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.

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

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

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

Listing 5.16: *ValidationEngine class builder pattern using method chaining (TypeScript)*

Accessing Data by Evaluating JSONPath Expressions

To be able to access essential information stored in the current runtime scope a validation process, a JSONPath expression can be used in certain attributes of a validation

---

[18]*Not skipped.*

[19]Please take a look into Figure A.2 to see a flow diagram for the validation process.

[20]Runtime secrets are used to store confidential information such as API keys.

[21]*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.

rule. The provided runtime information of a validation process includes the customer information, runtime secrets and during a *condition* evaluation, the HTTP response from an external endpoint.

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

```typescript
import jp from "jsonpath"

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

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

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

Listing 5.17: *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.

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

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

```
 7
 8    static setClient(context: Context) {
 9      this.context = context
10    }
11  }
12
```

Listing 5.18: *Usage of the singleton pattern and dependency injection in Agent class (TypeScript)*

Operators

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

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

```typescript
 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), // equals
 9    gt: new NumberOperator("gt", (a, b) => b > a), // greater than
10    gte: new NumberOperator("gte", (a, b) => b >= a), // greater than equals
11    lt: new NumberOperator("lt", (a, b) => b < a), // lesser than
12    lte: new NumberOperator("lte", (a, b) => b <= a), // lesser than equals
13  }
14
```

Listing 5.19: *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 on 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 by accessing the required data described by a JSONPath expression from the runtime information and running the operation defined by the particular condition's `type` and `operator` attributes.

```
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 5.20: *The usage of OperatorFactory class in the Evaluator class (TypeScript)*

A condition can either be a single condition, or multiple conditions, wrapped inside an *ANY* or *ALL* attribute. The evaluation of a single condition is different from the evaluation of multiple conditions. To facilitate the different logic of evaluating conditions, two subclasses of the `Evaluator` class is created. `ConditionEvaluator` evaluates a single condition, while the `BooleanConditionEvaluator` is responsible in evaluating multiple conditions and handling the logic behind evaluating the *ANY* and *ALL* modifier. To simplify the instantiation of the suitable `Evaluator` subclass, the `EvaluatorFactory` class is created, following the *Factory Method* design pattern described in [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 5.21: *EvaluatorFactory usage in ValidationEngine class (TypeScript)*

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

5.3.3. Notification on Suspicious Cases

In [15], Subramoni et al. describes an AMQP *exchange* as a routing mediator that copies and sends the message published by a message publisher to zero or more message queues. The FDS acts as the message publisher of the system and publishes a message to a pre-defined exchange on every completion of a validation process. An interface to access and publish a message to the exchange is implemented using the *Singleton* [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 the exchange used in the system is the *fanout* exchange. When a message is published to a fanout exchange, the message will be routed to all the queues bound to the exchange, which is ideal for the use case of the current notification system.

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

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

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

Listing 5.22: *Openning a connection to RabbitMQ instance (TypeScript)*

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

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

Listing 5.23: *Publishing a validation result to the RabbitMQ exchange (TypeScript)*

There can be many consumers consuming the messages published to the exchange, and the consumers can run certain actions regarding based on their internal logic. The message consumer can, for example email the concerned parties if the fraud score exceeds a certain threshold or even automatically block a customer if a specific rule

evaluation failed. To consume a message published to the exchange, a connection to the RabbitMQ instance should be opened, and a dedicated message queue (ideally created only for internal usage of the consumer itself) is required.

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

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

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

Listing 5.24: *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[22].

5.3.5. Validation Real Time Progress

The *Observer* [4, pp. 293-303] design pattern will be implemented to provide the functionality of a real time progress update of a running validation process. An `EventEmitter` is used here to subscribe to and publish certain events during a validation process. The FDS publishes a `validation_event:update` event whenever a rule evaluation is done, containing the validation ID attached to the event name and sending the current validation result on the payload. When the particular validation process is done, the FDS also publishes a `validation_done` event, containing the validation ID

---

[22]Runtime secrets are not cached.

on event name, as an indicator that the validation process is completed and no further events with the attached validation ID will be sent.

```
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 5.25: *Publishing events on certain validation events using the EventEmitter (TypeScript)*

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

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

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

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

```
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 }
24
```

Listing 5.26: *Writing to SSE stream when certain events are published (TypeScript)*

5.4. View

This chapter discusses the implementation of the features on the user interface described in subchapter 4.4.3.

5.4.1. Navigation

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

- Home page (path: /)
- Create a new rule (path: /rules/new)
- Edit a rule (path: /rules/:ruleName)[23]
- List of rules (path: /rules)
- Create a new validation (path: /validations/create)
- See validation progress for specific validation ID (path: /validations/:validationId)
- List of validations (path: /validations)

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

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

---

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

5.4.2. Rule Management Form

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

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

Conditions Section

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

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

```
const getAvailableOperators = (type: ConditionType) => {
  switch (type) {
    case "string":
      return [
        { label: "Equals", value: "eq" },
        { label: "Starts with", value: "starts" },
```

```
 7        { label: "Includes", value: "incl" },
 8        { label: "Ends with", value: "ends" },
 9      ]
10    case "number":
11      return [
12        { label: "Greater than", value: "gt" },
13        { label: "Greater than equals", value: "gte" },
14        { label: "Lesser than", value: "lt" },
15        { label: "Leser than equals", value: "lte" },
16        { label: "Equals", value: "eq" },
17      ]
18    case "array":
19      return [
20        { label: "Includes", value: "incl" },
21        { label: "Excludes", value: "excl" },
22        { label: "Number of items equals", value: "len" },
23        { label: "Is empty", value: "empty" },
24      ]
25    case "boolean":
26      return [{ label: "Equals", value: "eq" }]
27    default:
28      return []
29  }
30 }
31
```

Listing 5.27: *Function to get list of available operators based on a condition's type attribute (TypeScript)*

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

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

An additional validation is implemented in the CONDITIONS section. The validation not only make sure that all the required fields are filled, but also the value of the

fields itself. For example, the validation will throw an error if the Type field is set to "Number", but the input value of the Value field is not a valid numerical value.

Autocomplete Input

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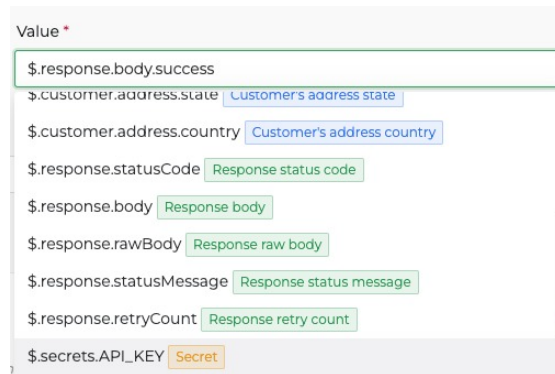

Figure 5.4.: *Screenshot of the autocomplete input usage*

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

- PATH field on CONDITIONS section
- VALUE field on CONDITIONS section
- Value field of a dynamic input[24]

5.4.3. Validation Form

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

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

Listing 5.28: *Prefilling form values with a sample customer data (TypeScript)*

---

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

Figure 5.5.: *Screenshot of the validation form*

When the user filled the fields properly and clicked the VALIDATE CUSTOMER button, the UI sends an HTTP POST request to the FDS with the customer data as the payload to schedule a new validation process. As the FDS returns an ID of the validation process, the UI will then redirect the user to the validation progress page, showing the progress of the particular validation process in real time.

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

```

Listing 5.29: *(TypeScript)*

5.4.4. Validation Progress

To receive the real time event stream sent by the FDS as described in subchapter 5.3.5, the `EventSource` API can be used on the client side. With the `EventSource` API, a persistent connection to the FDS will be opened, and the messages sent will be received by the client in real time. Because the FDS sends the messages as a string in an event stream, the UI has to parse the message content into a valid JSON object before processing it. It is also important to close the connection to the event stream when it's no longer needed. The logic of closing the open connection when user leaves the current page is implemented on the UI.

```typescript
const source = new EventSource(url)

source.onmessage = messageEvent => {
  try {
    const data = JSON.parse(messageEvent.data)
    // Do data processing
```

```
7    } catch {
8      // Handle error, if the message data is not a valid JSON object
9    }
10 }
11
```

Listing 5.30: *Using the EventSource API in the browser (TypeScript)*

As the content of the message is parsed and identified as a valid `ValidationResult` object, it is saved into the current state of the `Validation` component, which renders the events of a validation process into a timeline component to give the user a better graphical overview of the current process. Vue3 uses the *Proxy*[4, pp. 207-217] design pattern under the hood to establish a reactivity system on the UI, providing the possibility to update the timeline component every time a new validation result is published by the FDS.

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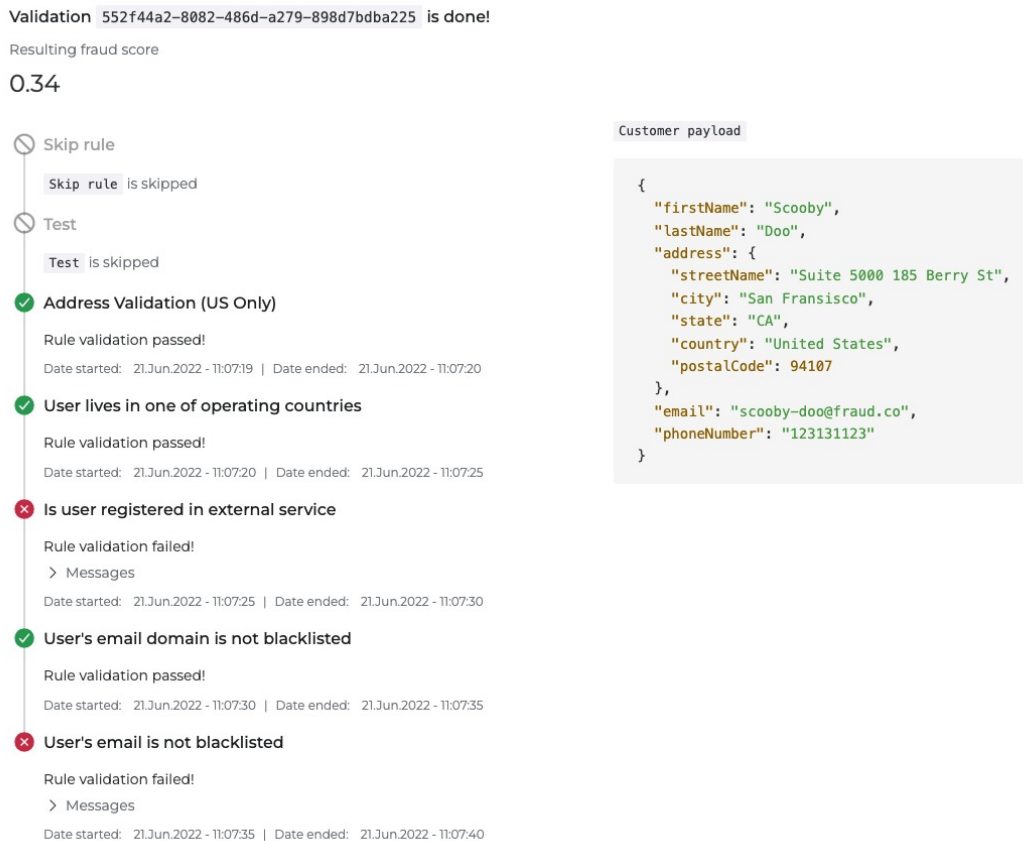

Figure 5.6.: *Screenshot of the validation progress in real time*

5.4.5. Runtime Secrets

A component to display a list of the available runtime secrets in a dialog is created. For security purposes, only the key of the secrets are displayed within the component. The component also provides a possibility to create a new runtime secret and to delete an existing secret. User can create a new secret by clicking on the CREATE A NEW SECRET button, and adding the essential information on the form fields displayed and delete an existing secret, a DELETE button is displayed next to each secret's key name. When a user creates a new runtime secret or deletes an existing secret, the autocomplete options are changed according to the latest list of keys available in the database.
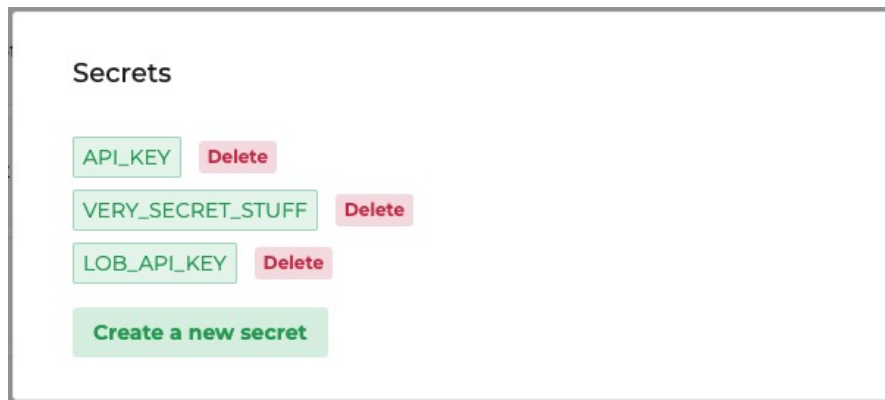
Figure 5.7.: *Screenshot of a component to list available runtime secrets*

6. Test

[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.

[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] 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).

[7] Loriot. *Möpse und Menschen. Eine Art Biographie. Zurich*. In: faz.net. Online: https://media0.faz.net/ppmedia/aktuell/feuilleton/1461387463/1.721778/format_top1_breit/die-steinlaus-trotzt-seit.jpg; letzter Zugriff: 13 VI 19. 1983.

[8] Loriot. *Steinlaus, Loriot Katalog, Diogenes Verlag, Zürich*. In: tagblatt.de. Online: https://www.tagblatt.de/Bilder/Loriots-legendaere-Steinlaus-Loriot-Katalog-1993-2003-125217h.jpg; letzter Zugriff: 14 VI 19. 1993, 2003.

[9] 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.

[10] 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.

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

[12] Pschyrembel online. *Steinlaus*. Online: https://www.pschyrembel.de/Steinlaus/K0LHT; letzter Zugriff: 14 VI 19. 2016.

[13] 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.

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

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

[16] Wikipedia. *Academic Use*. Online: https://en.wikipedia.org/wiki/Wikipedia: Academic_use; letzter Zugriff: 13 VI 19. 2019.

8. List of Abbreviations
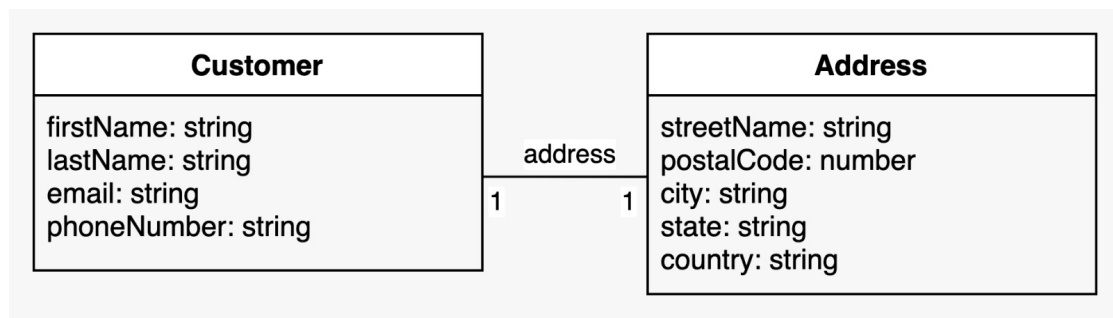
9. Glossary

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

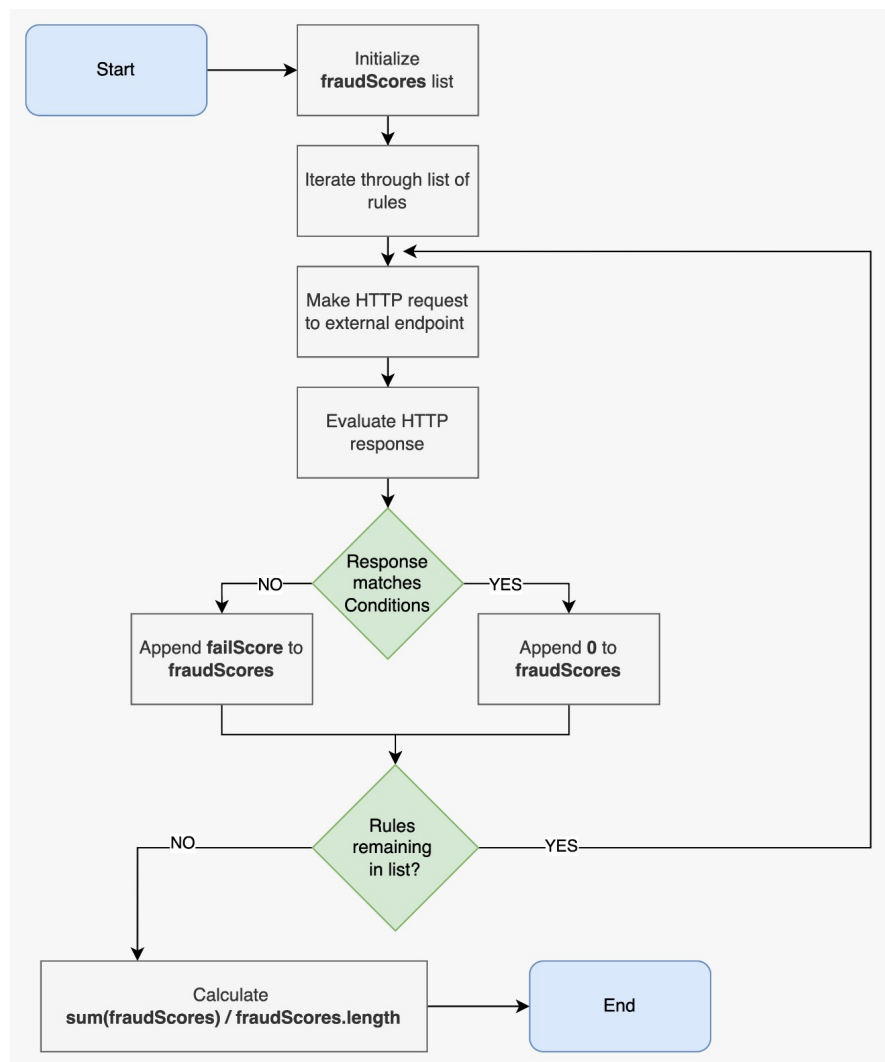- Achten Sie auf eine neutrale, fachliche Sprache. Keine „Ich"-Form.

A. Appendix

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

- Zitieren Sie zitierfähige und -würdige Quellen (z.B. wissenschaftliche Artikel und Fachbücher; nach Möglichkeit keine Blogs und keinesfalls Wikipedia[1]).
- 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[2], oder
  - Beim Lernzentrum[3].

---

[1]Wikipedia selbst empfiehlt, von der Zitation von Wikipedia-Inhalten im akademischen Umfeld Abstand zu nehmen [16].

[2]Z.B. [1], [2]

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

A. Appendix

- Nutzen Sie LATEX[4].

---

[4]Kein Support bei Installation, Nutzung und Anpassung allfälliger LATEX-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