Technical Design

Christopher Sulistiyo

Quality ICT B.V.

Christopher.sulistiyo@student.nhlstenden.com

ICT & IC Information Technology Department Emmen

Version 1.1 - 21/02/2024

Version Control

Version	Activities	Date
Initial version 1.0	Draft version	06/02/2024

Remarks

Any changes and new developments that have a significant impact on the project proceedings will be noted here.

Contents

Chapter 1

Introduction

1.1 Background and Context

This document serves as a comprehensive blueprint for the development and integration of this project. Therefore, this document will outline the technical specifications, architecture, methodologies, and implementation strategies necessary to achieve the project's objectives, as it is essential that every artefact is documented in the software development lifecycle to provide a detailed roadmap for future developers, project managers, and stakeholders for the making of new features for this project. Additionally, it ensures that all team members have a clear understanding of project's scope and its objectives in terms of software development, which is to outline the architecture, components, and implementation details of the SentinelOne integration into the QaaS App. Specifically, this document will:

- Provide an overview of the project's objectives, scope, overall architecture, and design principles guiding the integration process.
- Describe the project's architecture and components involved in the integration, including the client-side front-end built with Flutter, and the back-end powered by Firebase Cloud Functions and Node.js.
- Explain the workflow by illustrating the data flow and interaction between various components, ensuring seamless integration and real-time updates.
- Outline the implementation details and methodologies.
- Provide a detailed explanation of the database design.
- Discuss and highlight the security considerations, measurements and future improvements. It will further emphasize the security protocols and best practices implemented to maintain data integrity and privacy.
- Outline the development process by detailing the approach, including the use of design patterns and how it facilitates maintainability and scalability.

Chapter 2

System Requirements

System requirements are essential for the integration of SentinelOne EDR to the QaaS App. It serves as the foundation for the entire development process. It provides a clear understanding of what the system must achieve, guiding developers in creating a solution that meets the specified needs.

2.1 Client-Side Requirements

- Browser Compatibility: the web application should be compatible with modern web browser such as Google Chrome, Mozilla Firefox, Microsoft Edge, Opera, and Safari.
- Device Compatibility: the web application should be responsive and function correctly on various devices, including desktops, tablets, and smartphones.
- Minimum Screen Resolution: the application should support a minimum screen resolution of 1280x720 pixels.
- Performance: to ensure a smooth performance with minimal latency, even on devices with lower hardware specifications.
- Security: implement client-side security measures such as HTTPS, secure cookies, and Content Security Policy (CSP) to ensure data protection and security against common web vulnerabilities.
- Usability: design the UI to be intuitive and easy to navigate for non-technical users, incorporating accessibility standards.

2.2 Sever-Side Requirements

1. Firebase Cloud Functions:
   - Version: Use the latest stable version of gen 2.0 of Firebase Cloud Functions
   - Memory Allocation: at least allocate between 256 MB to 2 GB of memory per function, based on performance requirements.
   - Timeout Settings: set appropriate timeout settings for functions (default is 60 seconds), depending on the function complexity in case an infinite loop occurs or other unexpected behaviour.
   - Dependencies: include necessary Node.js dependencies, such as `firebase-admin`, `firebase-functions`, `axios`, and other project-specific packages.

Additionally, ensure proper logging, monitoring, and error handling mechanism are in place to maintain system reliability and quickly address any issues.

2.3 Integration Requirements

- SentinelOne API Integration:
  - Authentication: implement secure API authentication by using API key that have the appropriate access to fetch the wanted data.
  - Data Fetching: schedule regular intervals for fetching data from SentinelOne API to ensure up-to-date information is available in the application. In case of an error, do not show the technicality in the client instead make an appropriate user-friendly error page and message and redirect the user to that page.
  - Error Handling: Implement robust error handling for API calls to manage failures and retires gracefully. Also, document what was the source of the error on the Firebase function logs for better debugging.
- Firebase Authentication: ensure that the user is always logged and have necessary permission to access the SentinelOne dashboard of the application. Authentication can be implemented by using:
  - User Management: utilize Firebase Authentication for managing user sign-up, sign-in, and password recovery.
  - Role-Based Access Control: implement role-based access control to restrict access to sensitive data and actions.
- Database:
  - Firestore: use Firebase Firestore for real-time data storage and retrieval.
  - Data Security: implement Firestore security rules to control access to the data.

2.4 Database Requirements

- Database Type: use a NoSQL database such as Firebase Firestore for storing user data, logs, and other relevant information.
- Data Structure: design a schema that efficiently stores and retrieves data, ensuring data consistency and integrity.
- Security: implement database security measures such as encryption, access control, and data validation to protect against unauthorized access and data breaches.
- Scalability: design the database to scale horizontally to accommodate increasing data volume and user traffic.

2.5 Deployment Requirements

- Deployment Environment: deploy the application on a cloud platform, in this project's case Firebase Hosting.
- Continuous Integration/Continuous Deployment (CI/CD): set up CI/CD pipelines to automate the deployment process and ensure consistency across environments.
- Monitoring: implement monitoring tools on the said cloud provider such as Firebase Performance Monitoring to track system performance, detect issues, and optimize resource utilization.

2.6 Non-Functional Requirements

- Scalability: design the system to scale horizontally to handle an increasing number of clients and devices.
- Reliability and Availability: ensure high availability of the system with minimal downtime. Implement robust error handling and recovery mechanisms to maintain system reliability.
- Security: protect sensitive data using encryption in transit and at rest. Using Google Secret Manager to store sensitive information such as API keys and credentials where those data will be encrypted in the cloud. Do not include those in the front-end or back-end code to avoid unintended exposure.
- Documentation: provide comprehensive documentation for developers, administrators, and end-users to understand the system architecture, functionality, and usage.
- Testing: conduct thorough testing, including unit tests, integration tests, and end-to-end tests, to ensure the application functions as expected and meets the specified requirements.
- Compliance: ensure the application complies with relevant regulations and standards, such as GDPR, HIPAA, and PCI DSS, to protect user data and maintain legal compliance.
- Support: provide ongoing support and maintenance to address issues, implement updates, and enhance the application's functionality based on user feedback.
- Maintainability: follow the web architectural design patterns like MVC, MVVM, or MVP to ensure codebase maintainability. Document the code and APIs thoroughly in a README file or code commenting for future developers.
- Usability: conduct usability testing to ensure the application is easy to use for non-technical clients. Provide help and support documentation such as User Manual and/or Developer Manual within the application.
- Compability: ensure the Flutter application works on every screen including mobile screens like Android, iOS, and tabloid devices. Ensure compatibility with various web browsers for the web-based components.

Chapter 3

SentinelOne API Architecture

Before diving deeper into the project’s architect itself, it is important to understand what kind of API does SentinelOne provides to its audience and how SentinelOne structures its API.

JSON format JSON is a lightweight data-interchange format that is easy for humans to read and write while being also easy for machines to parse and generate too. It is based on a subset of the JavaScript programming language and is commonly used for representing structured data. JSON is often used for transmitting data between a server and a web application, as well as for storing configuration data.

JSON data is organized into key-value pairs, where keys are strings and value can be strings (enclosed in double quotes ‘””‘), booleans (‘true‘ or ‘false‘), numbers, objects (unordered collections of key-value pairs enclosed in curly braces ”), arrays (ordered collections of values enclosed in square brackets ’[]’), or null.

Listing 3.1: Example of a JSON response

```
{
      "name": "John Doe",
      "age": 30,
      "isStudent": false,
      "cars": [
           {"name": "Ford", "models": ["Fiesta", "Focus", "Mustang"] },
           {"name": "BMW", "models": ["320", "X3", "X5"] },
           {"name": "Fiat", "models": ["500", "Panda"] }
      ],
      "hobbies": ["Reading", "Gaming", "Traveling", "Cooking", "Photography",
         "Painting", "Gardening"],
      "address": {
           "street": "Sesame Main Street",
           "city": "New York",
           "zip": "10001"
      }
}
```

REST API: if SOAP is like an envelope, RESTis like a more lightweight postcard. REST APIs are considered the gold standard for scalability and are highly compatible with microservice architecture. It is the often used protocol in the context of building APIs for web-based applications. REST itself is not a protocol, but an architectural style for designing networked applications, defining a set of constraints and principles that define how web services should be structured and interact with each other.

APIs that follow REST principles are called RESTful APIs. The are RESTful as long as they comply with the 6 guiding constraints of a RESTful system:

- **Client-server architecture**: the architecture is composed of clients, servers, and resources, and it handles requests through HTTP.
- **Statelessness**: no client is stored on the server between requests. The server should process and complete each request independently. Information about the session state is, instead, held with the client. The clients can do this via query parameters, headers, URIs, request body, etc.,
- **Cacheable**: simply, the clients should be able to determine whether this response is cacheable from their side, and if so, for how long. If a response is cacheable, the client has the right to return the data from its cache for an equivalent request and specified period, without sending another request to the server. A well managed caching mechanism can eliminate the need for some client-server interactions
- **Layered system**: client-server interactions can be mediated by additional layers. These layers could offer additional features like load balancing, shared caches, or security.
- **Uniform interface**: this is the core to design RESTful APIs. There should be a uniform and standard way of interacting with a given server for all client types. The uniform interface helps to simplify the overall architecture of the system. This includes 4 facets:
  - Resource identification in request: resources are uniquely identified in requests and are separate from the representations that are returned to the client using URI.
  - Resource manipulation through representations: clients receive files of a uniform that represent resources. These representations must have enough information to allow modification or deletion of the resource's state in the server, as long as they have the required permissions.
  - Self-descriptive messages: each message returned to a client contains enough information to describe how the client should process the information further, such as additional actions that can be performed on the resource.
  - Hypermedia as the engine of application state: after accessing a resource, the REST client should be able to discover through hyperlinks all other actions that are currently available.
- **Code on demand (optional)**: servers can extend the functionality of a client by transferring executable code.

REST APIs are high-performing (especially over HTTP2), time-tested, and support many data formats. They also decouple the client and server, making sure of independent evolution. However, building a true REST API is difficult because it requires a disciplined adherence to the Uniform Interface constraint. Some organizations trade off the long-term benefits of a truly REST API for other HTTP API protocols that have similar benefits but adhere to REST constraints more liberally. REST requests typically include these key components:

- **Endpoint**: the uniform resource identifier that locates the resource on the internet is part of this component. URLs are the most common type of URI.
- **HTTP Method**: this component outlines the four basic processes that a resource can be subjected to: POST (create a resource), GET (retrieve a resource), PUT (update a resource), and DELETE (remove a resource).
- **Headers**: data related to the server and the client are stored in this component. Like in SOAP, one can also use REST headers to store authentication measures such as API keys, server IP addresses, and the response format.

- **Body**: this component contains additional information for the server, such as data that needs to be added or replaced.

Listing 3.2: Different HTTP methods in REST

```
GET /users Retrieve list of all users
GET /users/{id} Retrieve details a specific user by their ID
POST /users Create a new user
PUT /users/{id} Update a specific user by their ID
DELETE /users/{id} Delete a specific user by their ID
```

Listing 3.3: REST's Example Request

```
GET https://api.example.com/users/123
```

There are basically 5 main components of the data that are useful for the user to see in the QaaS App:

3.1 Endpoints

Endpoints within an organization's network that have SentinelOne Agents installed will have their information collected and analysed. This various types of information plays a vital role in maintaining the security integrity of the network. Some overview about an endpoint that SentinelOne gathers and analyses are:

1. **System Information** :
   (a) **Endpoint Name**: the name assigned to the device.
   (b) **IP Address**: the unique identifier assigned to the device on the network.
   (c) **MAC Address**: the unique identifier assigned to the device on the network.
   (d) **Operating System**: details about the OS version and type (Windows, macOs, Linux, etc.).
   (e) **Hardware Specifications**: information about the CPU, memory, disk space, and other hardware details.
   (f) **Last Seen**: the last time the endpoint was active and communicated with SentinelOne Console.
   (g) **Agent Version**: the version of SentinelOne Agent installed on the endpoint.
   (h) **User Account**: The user account of the endpoint.
   (i) **Device Type**: the type of device (e.g., desktop, laptop, server, workstation, mobile device, IoT device, etc.,).
2. **Security Status**:
   (a) **Agent Status**: whether the Agent is active, inactive, decommissioned or needs an update.
   (b) **Health Status**: the overall health of the endpoint from a security perspective.
   (c) **Threat Level**: any current threats and their severity.
3. **Performance Metrics**:
   (a) **CPU Usage**: the percentage of CPU resources being used by the endpoint.
   (b) **Memory Usage**: the percentage of memory resources being used by the endpoint.
   (c) **Disk Usage**: the percentage of disk space being used by the endpoint.

4. **Response Actions**:
    (a) **Quarantine Status**: information on files or processes that have been quarantined.
    (b) **Mitigation Actions**: actions taken by SentinelOne to mitigate security threats (e.g., killing processes, rolling back changes).
5. **Application Information**
    (a) **Installed Applications**: list of applications installed on the endpoint, including versions and publishers.
    (b) **Running Processes**: real-time information on running processes and their resource usage.
    (c) **Active Sessions**: current sessions associated with user accounts, indicating who is actively logged in.
6. **Activity Logs**:
    (a) **Process Activity**: logs of processes that have run on the endpoint, including those flagged as suspicious.
    (b) **Network Activity**: details of network connections made by the endpoint, including potentially malicious connections.
    (c) **User Activity**: information about user logins and actions that could be relevant for security monitoring.
7. **Endpoint Configuration**:
    (a) **Installed Software**: list of software installed on the endpoint.
    (b) **Running Services**: services running on the endpoint.
    (c) **Startup Programs**: programs configured to run at startup.
8. **Threat Data**:
    (a) **Detected Threats**: information on threats detected on the endpoint, including malware, exploits, and suspicious activities.
    (b) **Threat Details**: specific information about each threat, including type, severity, and status (e.g., quarantined, mitigated).
    (c) **Threat Timeline**: the history of threat detection and mitigation actions taken on the endpoint.
9. **User Information**
    (a) **User Accounts**: list of user accounts active on the endpoint, including their names, roles, groups, and privileges.
    (b) **Login Activity**: records of login attempts, successes, and failures, along with timestamps.
    (c) **User Active Sessions**: list of current sessions associated with user accounts, indicating who is actively logged in.
10. **Policy Compliance**:
    (a) **Policy Adherence**: whether the endpoint adheres to the security policies configured in SentinelOne. The data is in *boolean* format.
    (b) **Policy Violations**: detected deviations or violations of security policies, indicating areas of concern.
11. **File and Content Information**
    (a) **File Hashes**: unique identifiers for files, useful for detecting known malicious files.
    (b) **File Types**: information on the types of files being accessed or transferred.

    (c) **Content Analysis**: textual content of documents and e-mails, scanned for signs of phishing, malware, or other threats.

12. **Behavioural Analytics**
    (a) **Anomaly Detection**: identifies unusual patterns of behaviour that could indicate a security issue.

    (b) **User Behaviour Baseline**: established normal behaviour patterns for users and devices, aiding in anomaly detection.

    (c) **Device Reputation**: assesses the safety and reputation of devices based on their behaviour and interactions.

13. **Threat Intelligence**
    (a) **Threat Reports**: incorporates external threat intelligence feeds to enrich the detection capabilities.

    (b) **Signature Updates**: regular updates to malware signatures and indicators of compromise (IOCs).

14. **Security Events**
    (a) **Alerts**: generated notifications of potential security issues, requiring investigation and response.

    (b) **Incident Logs**: detailed records of security incidents, including timestamps, affected systems, and actions taken.

15. **Compliance and Policy Enforcement**
    (a) **Policy Violations**: detected violations of organizational security policies.

    (b) **Audit Trails**: records of policy enforcement actions and compliance checks.

16. **Hardware Information**:
    (a) **Device Model**: the model and specifications of the hardware.

    (b) **Device Serial Number**: the unique identifier assigned to the device on the network.

    (c) **BIOS/UEFI**: information about the BIOS and UEFI firmware versions.

3.2 Threats

Threats represent all the previous types of cyberattacks that have happened on endpoints that are connected in a single network, therefore forming a site group in SentinelOne Console. All the threats that are listed have been detected by SentinelOne agents, therefore they no longer possess any danger to the environment. These threats are categorized on their behaviour and potential harm they cause to the endpoints. It is part of the SentinelOne capabilities to provide comprehensive visibility and control over the security of endpoints within an organization's network. Some key threat categories that SentinelOne addresses are:

1. **Malware**
    (a) **Virus**: self-replicating code designed to cause damage to a computer system.

    (b) **Trojan Horse**: malicious software disguised as legitimate software.

    (c) **Worms**: autonomous program that replicate themselves by modifying the host program.

    (d) **Spyware**: software designed to gather personal or confidential information without consent.

    (e) **Ransomware**: encrypts files on a victim's computer or network, demanding a ransom payment for the decryption key.

2. **Adware and PUPs (Potentially Unwanted Programs)**: are software that automatically delivers advertisements, often bundled with free software downloads. They are usually not malicious but unwanted by the user.
3. **Exploits**: are pieces of code specifically crafted to take advantage of a bug or vulnerability in software or hardware to gain unauthorized access or privileges. It can be broken down into 2 types:
   (a) **Zero-Day Attacks/Exploits**: attacks that exploit unknown vulnerabilities in software or hardware to those who should be interested in mitigating them, including developers and users.
   (b) **Known Vulnerabilities**: attacks that exploit known vulnerabilities in software or hardware that have been disclosed to the public, but have not been patched or fixed by the vendor.
4. **Phishing and Credential Theft**: involves sending fraudulent communications, usually e-mails, to trick recipients into revealing sensitive information, such as passwords and credit card numbers.
5. **Botnets**: are networks of compromised computers that are controlled by malicious actors, often being used to perform DDoS attacks, send spam, or steal data.
6. **Advanced Persistent Threats (APTs)**: are long-term targeted attacks on specific systems, often state-sponsored, aimed at stealing intellectual property or sensitive information.
7. **Fileless Malware**: malicious malware that do not rely on files and instead work directly in computer's memory and exploit existing legitimate tools or scripts on the system (e.g., PowerShell, WMI)
8. **Insider Threats**: come from individuals within an organization who have access to sensitive information, either intentionally or unintentionally causing harm.
9. **Network Attacks**:
   (a) **Man-in-the-Middle (MitM)**: an attack where the attacker secretly intercepts and possibly alters the communication between two parties.
   (b) **Denial-of-Service (DoS)**: an attack that attempts to make a machine or network resource unavailable to its intended users.
10. **Suspicious Activities**
    (a) **Suspicious Processes**: unusual or potentially harmful processes running on the system.
    (b) **Suspicious Network Traffic**: unusual outbound or inbound network traffic that may indicate malicious activities.
11. **Rootkits and Bootkits**: are malware designed to gain unauthorized root or administrative access to a system and hide its presence.
12. **Insider Threats**: are threats that come from individuals within an organization who have access to sensitive information, either intentionally or unintentionally causing harm.

The types of information regarding the threat that can be provided by SentinelOne for analysing and responding to these security incidents are the following:

1. **Threat Overview**
   (a) **Threat Name**: the name or type of the detected threat (e.g., Trojan, Ransomware, Phishing Attack).
   (b) **Detection Source**: the method or technology used to detect the threat (e.g., behavioural AI, signature-based detection, threat intelligence).
2. **Threat Details**

   (a) **Severity Level**: the assigned severity of the threat (e.g., low, medium, high, critical).
   (b) **Status**: the current status of the threat (e.g., detected, mitigated, undetected).
   (c) **Detection Time**: the time at which the threat was detected.

3. **Threat Indicators**
   (a) **Indicators of Compromise (IOCs)**: specific attributes associated with the threat, such as file hashes, IP addresses, URLs, and domain names.
   (b) **Malicious File Details**: information about malicious files, including file path, file name, file hash, and file size.

4. **Threat Behaviour**
   (a) **Process Information**: details of the process involved in the threat, such as process names, IDs, parent processes, and command-line arguments.
   (b) **Execution Flow**: the sequence of actions taken by the threat, including process creation, file modifications, registry changes, and network connections.
   (c) **Behavioural Indicators**: behavioural patterns observed during the threat's execution, such as attempts to evade detection, persistence mechanism, and system modifications.

5. **Affected Endpoints**: endpoint details as was already described in (*1*)

6. **Mitigation and Response Actions**:
   (a) **Quarantine Actions**: information on whether the malicious file or process has been quarantined.
   (b) **Killed Processes**: details of any processes that were terminated as a part of the mitigation efforts.
   (c) **Rollback Actions**: if supported, information on any rollback actions taken to revert the system to its pre-infected state.
   (d) **Manual Interventions**: details of any manual actions taken by admins, such as isolating the endpoint or initiating a deep scan.

7. **Threat Timeline**
   (a) **Detection Events**: a chronological list of events related to the detection of the threat.
   (b) **Mitigation Events**: a timeline of mitigation actions taken by SentinelOne and admins.

8. **Threat Analysis**
   (a) **Threat Classification**: classification of threat based on its behaviour and characteristics (e.g., file-based, fileless, memory-based) as was listed above.
   (b) **Threat Intelligence**: additional context from threat intelligence sources, such as associations with known threat actors or campaigns.

9. **Clarification**
   (a) **Attack Storyline**: a visual representation of the threat lifecycle and its activities within the endpoint environment.
   (b) **Process Tree**: A visual depiction of the process hierarchy and how the threat interacted with other processes.

10. **Contextual Information**:
    (a) **Related Threats**: information about other threats that are similar or related to the detected threat.

(b) **Threat Evolution**: insights into how the threat has evolved over time and any changes in its behaviour or tactics.

Additionally in the API, threats can also be filtered from:

Analysis Verdict (performed by user, marking it as)	Threat Mitigation Status	Incident Status	A.I. Confidence level
• False positive • Suspicious • True positive • Undefined	• Mitigated • Not mitigated • Marked as benign	• Resolved • Unresolved • In progress	• Malicious • Suspicious • N/A

Table 3.1: Different filters that can be applied to the Incidents page #1

Engine	Cloud Provider	OS	Classification
• SentinelOne Cloud • On-Write Static A.I. - Suspicious • Behavioral A.I. • On-Write Static A.I. • Reputation • Cloud Detection • User-Defined Blocklist • Documents, Scripts • Anti Exploitation / Fileless • Intrusion Detection • Potentially unwanted application • Lateral Movement • Remote Shell • Manual • Application Control • Threat Intelligence • Watctower • Driver Blocking	• Azure • AWS • GCP • OCI • ESXi	• Windows • Linux • Mac • Windows Legacy	• Malware • PUA • Virus • Infostealer • Hacktool

Table 3.2: Different filters that can be applied to the Incidents page #2

Initiated by	Time	Free text search

• Agent Policy	• Recent	• Content Hash
• Full Disk Scan	• Last 24 Hours	• Cloud Account
• Local agent command	• Today	• Cloud Image
• Deep Visibility Command	• Last 48 Hours	• Cloud Instance ID
• Management console API	• Last 7 Days	• Cloud Instance Size
• On-Demand Scan	• Last 30 Days	• Cloud Location
• Custom Rule	• This Month	• Cloud Network
• Custom Alert	• Last 2 Month	• AWS Role
• Cloud Detection	• Last 3 Month	• AWS Security Groups
• Threat Intelligence	• Last Year	• AWS Subnet IDs
• Watctower	• Custom Range	• Azure Resource Group
		• GCP Service Account
		• Threat Details: e.g., ”This is a non-Microsoft binary that masquerades as a Microsoft executable”
		• File Path: e.g.,: ”\Device\HarddiskVolumeA\Users\Chris\Downloads\malicious.exe”
		• Endpoint Name: e.g., ”LT-Christopher”, ”LT 10-08”
		• UUID: e.g.,: ”0ff2a3409f284776a23432e9f6894afa”
		• Agent Version (at detection) e.g.,: ”23.1.4.650”
		• Agent Version (current)
		• Domain (at detection time)
		• Command Line Arguments
		• Initiated By (username): e.g., ”LuukAdmiraalMKBIT”
		• Storyline
		• Originated Process
		• Cluster Name
		• Node Name
		• Namespace Name
		• Namespace Labels
		• Controller Name
		• Controller Labels
		• Pod Name
		• Pod Labels
		• Container Name
		• Image Name
		• Container Labels
		• External Ticket ID
		• Node Labels

Table 3.3: Different filters that can be applied to the Incidents page #3

3.3 Ranger

Ranger in SentinelOne is an advanced network discovery and visibility feature designed to identify unmanaged and potentially vulnerable devices within an organization's network. It is showing information more about the network of the organization. It therefore extends SentinelOne's capabilities beyond endpoint protection, providing comprehensive visibility into the entire network environment. Some data that can be gathered are:

1. **Device Discovery**: information that are already discussed in (*1*).
2. **Network Visibility**:
   (a) **Network Segments**: information about the network segments or subnets where the devices are located.
   (b) **Connection Details**: information about the network connections, such as connected switches, routers, and access points.
   (c) **Communication Patterns**: details about the communication patterns of the device, including inbound and outbound connections.
3. **Device Inventory**:
   (a) **Firmware Version**: the version of the device's firmware, if applicable.
   (b) **Hardware Details**: information about the device's hardware, such as manufacturer, model, and serial number.
4. **Security Posture**:
   (a) **Managed vs. Unmanaged**: identification of whether device is managed (i.e., has the SentinelOne Agent installed). The data is in *boolean* format.
   (b) **Vulnerability Assessment**: details about known vulnerabilities associated with the device, based on its OS, firmware, or software.
5. **Policy Compliance**: information that are already discussed in (*10*).
6. **Device Behaviour**:
   (a) **Behavioural Analysis**: insights into the device's behaviour, including any suspicious or anomalous activities.
   (b) **Risk Assessment**: the risk level associated with the device based on its behaviour and security posture.
7. **Response Capabilities**:
   (a) **Mitigation Actions**: informa
   (b) **Notification Alerts**: alerts and notifications related to the device's security status, generated for any discovered vulnerabilities or suspicious activities associated with the device.
8. **Device Inventory Management**:
   (a) **Agent Deployment**: options to deploy SentinelOne agent to unmanaged devices directly from the console.
   (b) **Device Grouping**: ability to group devices based on criteria such as device type, location, or risk level for better management.
9. **Activity Logs**:
   (a) **Discovery Logs**: logs of discovery events, including timestamps and the nature of each event.
   (b) **Historical Data**: historical records of the device's presence on the network and any changes in its status.

10. **Reporting** :
    (a) **Reports and Dashboards**: customizable reports and dashboards that provide summaries and insights into the network's security posture and device inventory.
    (b) **Network Maps**: connection representations of the network, showing the relationships and connections between devices.

Unfortunately for this project scope, Q-ICT has turned off the Ranger feature, so this will be out of scope. But this document can then serve as a future blueprint in case they want to implement it later on the application.

3.4 Rogues

Rogues in SentinelOne is unmanaged devices or endpoints that have been detected on the network but do not have the SentinelOne Agent installed. Identifying rogue devices is crucial for maintaining network security and ensuring comprehensive endpoint coverage. The information shown about rogues are the following:

1. **Device Identification, Type and Classification**: containing the data that has been discussed in (*1*).
2. **Network Information**:
    (a) **Network Segment**: the network segment or subnet where the device is located.
    (b) **Connected Access Point**: information about the access point or switch the device is connected to.
3. **Discovery Details**:
    (a) **Discovery Date**: date and time the device was discovered on the network.
    (b) **Last Seen**: the last time the device was active on the network.
4. **Security Posture**:
    (a) **Risk Assessment**: an evaluation of the potential risk posed by the rogue device based on its behaviour and security posture.
    (b) **Vulnerability Detection**: information about any known vulnerabilities associated with the device's OS or software.
5. **Behavioural Analysis**:
    (a) **Suspicious Activities**: details of any suspicious activities or behaviours observed on the rogue device.
    (b) **Communication Patterns**: information about the device's communication patterns, including inbound and outbound connections.
6. **Policy Compliance and Violations**: information that are already discussed in (*10*).
7. **Mitigation and Response Actions**:
    (a) **Isolation Options**: options to isolate the device from the network to prevent potential threats.
    (b) **Agent Deployment**: options to deploy Agent to the rogue device for better management and security.
    (c) **Manual Actions**: records of any manual actions taken by admins, such as initiating a scan or quarantining the device.
8. **Reporting**: data that have been discussed in (*10*).
9. **Historical Data**

(a) **Past Incidents**: information on any previous incidents involving the rogue device.

(b) **Activity History**: historical activity logs for the device, showing patters over time.

Rogues are very closely related to Ranger, but the difference is that Ranger is more about the network itself, while Rogues are more about the devices. Because the Ranger functionality is turned off, this project scope will also not include Rogue data to the application.

3.5 Miscellaneous

This is the optional functionality provided by MGMT console, within it the user can see the following:

1. **Live Feed News**: the cybersecurity the latest news from SentinelOne.
2. **Customizable Notes**: customizable notes that can be added to the application by the users.

Chapter 4

Database Design

NoSQL Database

is a category of DB that provides a mechanism for storage and retrieval of data that is modelled in ways other than the tabular relations used in RDBMS. NoSQL DBs are typically designed to handle large volumes of unstructured or semi-structured data, such as JSON, XML, or binary objects, and they offer a flexible data model that can evolve over time. Both databases in Firebase, Firestore, and Real-time Database are NoSQL databases. Some characteristics of NoSQL DBs include:

- Data Model: NoSQL uses dynamic schema, which allows data to be inserted without having to define the schema first, while SQL uses a fixed schema to store data.
- Data Structure: NoSQL DBs use a variety of data structures such as key-value pairs, document-oriented, graphs databases, and columns-oriented; unlike SQL which only uses tables to store data.
- Querying: NoSQL uses variety of query languages, such as MongoDB query language or Cassandra's CQL, while SQL databases only use SQL as the unified language to query data.
- Data Consistency and ACID Compliance: databases that are ACID compliant means that they follow a set of rules to ensure that database transactions are processed reliably and securely. SQL databases are a good example of this, while NoSQL databases sacrifice some ACID properties to achieve higher performance and scalability.
- Use Case: NoSQL DBs are ideal for applications that need to handle large volumes of data and require high scalability and flexibility, such as social media platforms, real-time analytics, and content management systems.
- Scalability: NoSQL databases are traditionally designed for horizontal scalability, meaning they can easily scale out multiple servers or clusters to handle volumes of data and high throughput, which is more cost-effective and allows for better scalability. They are well-suited for distributed and cloud-based environments. SQL databases are traditionally designed for vertical scalability, meaning a single server is scaled up with more powerful hardware and power (CPU, RAM) to a single server. This can become expensive and limit scalability.

4.1 How Collection is Stored

Before diving into how the database is designed, a basic overview of the terminologies of Firestore is needed to understand the basis of the hierarchy of the database.

1. Collection: a group of documents that have the same structure. It is equivalent to a table in a relational database. Collections can contain multiple documents, and each document can contain multiple fields.

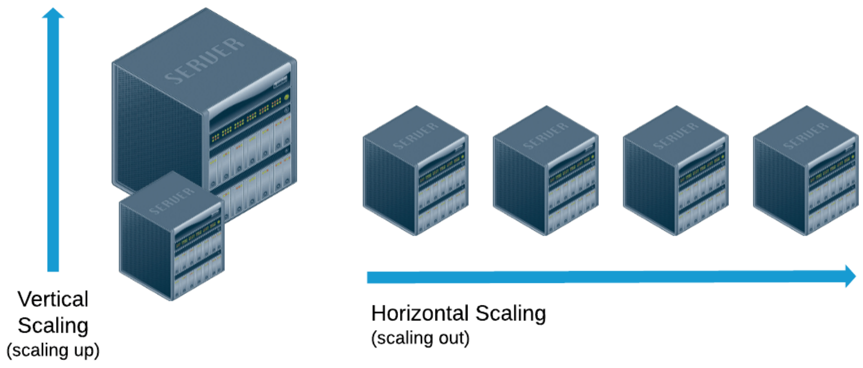


Figure 4.1: Vertical (SQL) VS Horizontal (NoSQL) Scaling

2. Document: refers to a single record in a collection. A document is essentially a JSON object stored in a collection. This JSON object can contain various fields, which are represented as key-value pairs. Each document in a collection can have a different structure. This is because Firestore is schema-less, meaning that it allows a flexible schema, and the data can be stored without any restrictions. Each document is identified by a unique ID, which is automatically generated by Firestore or can be set manually by the user. The document ID is used to uniquely identify the document in the collection.
3. Field: each document will then contain fields, which are essentially key-value pairs that hold the data for that document. Fields will have different types of data, such as the following:
   - String: the most common type of data. It can hold any text data, including numbers and special characters.
   - Number: can hold any numerical data. It can be integer or float.
   - Boolean: can hold 2 values which are true or false value.
   - Map: a collection of key-value pairs. It can hold any data type listed in this list.
   - Array: a collection of data of any type. It can hold any data type listed in this list.
   - null: the absence of any value.
   - Timestamp: a date and time value. It will be in the form of number of milliseconds since January 1, 1970 (Unix Epoch).
   - Geopoint: a geographical location.

Because Cloud Firestore is a NoSQL database, a different approach is needed to managing data compared to traditional SQL databases. It is built for automatic scaling, while still offering high performance and ease of application development. NoSQL is also schema-less, meaning that it allows a flexible schema, the data can be stored without defining the structure first. NoSQL databases are also scaled horizontally, by adding more servers to distribute the workload, instead of adding more power to a single server. Firestore NoSQL does in fact however, allow indexing to the database. There are 2 ways of how the author approach indexation for this project:

1. Document ID: is the usual form of indexation. Document IDs can be automatically generated by Firestore or generated manually by the user, keeping in mind that each document ID must be unique to each other.
2. Field ID: is the form of indexation that can be created by the user. The document itself is using a JSON format, and the user can create an index by making a specific field in the JSON document format, and then query it by making a filter to that field. This is useful for the author to store the user preferences in the Firestore database.

4.2 Data Types

- Endpoints: contains comprehensive set of information related to each endpoint (desktops, laptops, servers) that is being monitored by SentinelOne platform. This data will help administrators and security teams to manage, monitor, and respond to security events effectively.
- Threats: the list of malware that have been detected on endpoints that are connected to a specific network. The type of data that can be gathered are already discussed in 3.2
- Applications: contains information about outdated applications that can pose security risks to an endpoint according to the CVEs regularly posted by MITRE and NVD.
- Miscellaneous: contains SentinelOne news feed article and customizable notes.

4.3 Collection Types

There are 2 types of collection that are used throughout the project. These types were created by the author for the purpose of storing the required data for the application.

1. SentinelOne Data: containing the data from fetched from SentinelOne API, each unique to their respective Company Site ID. The data is further classified into types like what was discussed above (*4.2*). This collection will store its data in different way than in SQL. A new document will be created for each data model of the respective JSON response, and an indexation field will be created that will be based on the Site ID relative to the company. Firestore itself will determine the Document ID, to ensure that each document is unique.
2. User Preference: containing the data from each individual users about their choice of graphic types per widgets, types of data that wanted to be displayed, editing the widget title, table header columns customization, and pagination. This collection will store its data in the same way in SQL. Document IDs will be determined based on user ID, for the patency of Cloud Functions being able to determine which data from which user is being fetched. It is therefore easier to add, edit, delete fields of specific documents. The User Preference Collection also has a defined structure for each of its document, which mean they will contain the same fields that will be discussed later below.

4.4 Index Types

Moreover, indexes play a huge role in the database. Indexes are used to identify which data belongs to which, so that the Cloud Functions can know exactly what to fetch to a specific customer. Indexes also allow for faster search result if the request requires filtering of some data. There are 3 types of indexes that are used in the Firestore database:

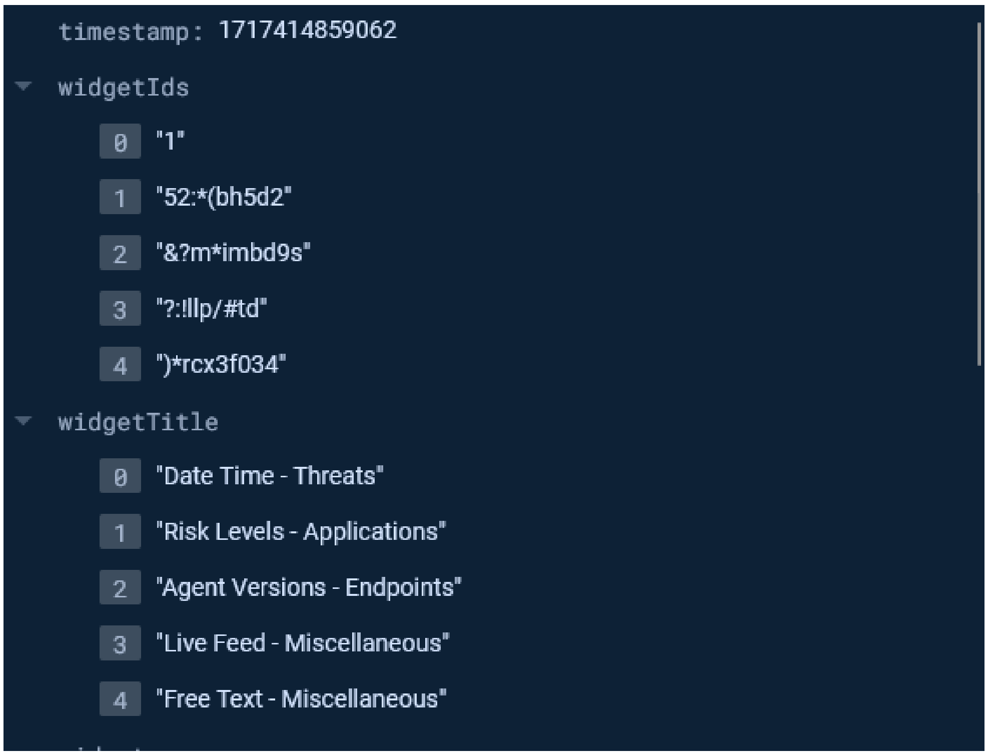
- Site ID: is mainly used for indexing the SentinelOne data. The purpose of storing the site ID in the field of a document is to differentiate data across different organizations that are part of Q-ICT's SentinelOne environment (the customers). This separation of data is mainly has got to do with the fact that a customer from a specific organization cannot see SentinelOne data outside of their own organization.
- User ID: is mainly used for the collections that store the user preferences. The main purpose of this index is to further differentiate the data between users of the same organization as users will have different customization to their widgets between each other.
- Data ID: this indexation is used to further differentiate certain varieties of SentinelOne data in Firestore. This is because each data can contain an additional model data specific to that data ID. For example, each threat that have

happened have their own timeline about what processes were undertaken by SentinelOne to mitigate the threat. This will then cannot be generalized by just the Site ID, but instead need more specific indexation.

Making of User Preference Table

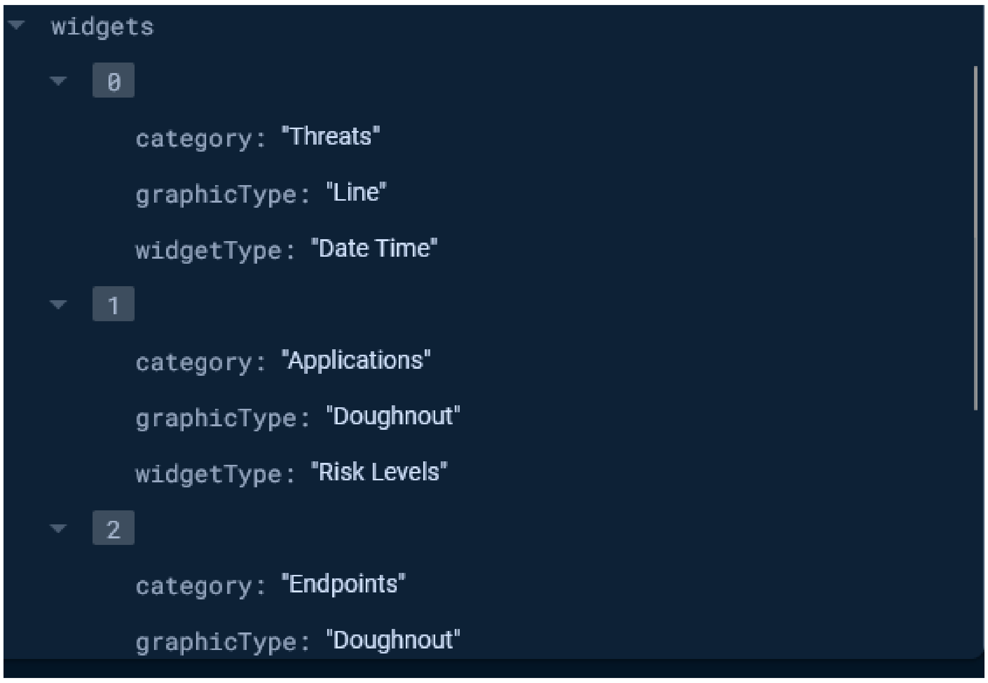
A special table in Firestore is created to store the user's preference, each document' ID is the user's ID, therefore ensuring uniqueness. In a document, there are 3 different fields:

- timestamp: containing date time in the format of timestamp. It is used as a reference for the Cloud Function to know when the data is last updated, and if it exceeds a certain threshold, the Cloud Function will fetch the data from the SentinelOne API again. This will be the field that the fetching cloud function needs to do its if-check first to determine if the data is outdated.
- widgetIds: containing an array of strings, each string is the ID of the widget that the user wants to see in the dashboard. This is used for the convenience of adding, updating, and deleting widgets.
- widgetTitle: containing also an array of strings, which serves as the title that the user chooses for their widgets. A widget title can have the same name as another widget, if the widget ID is different.
- widgets: containing a map of 3 different fields:
  - category: containing a string, which is the category of the widget. The category is used to group the widgets in the dashboard, so the user can easily find the widgets that they want to see. Currently, there are 3 categories: "Threats", "Endpoints", "Applications", and "Miscellaneous". In the future, additional 2 categories can be added: "Ranger (Network Discovery)" and "Rogues". Ranger is a feature that is used to discover the network of the client's machine, and Rogues is a feature that is used to detect any unauthorized devices that are connected to the client's machine. These 2 features are currently deactivated in the Q-ICT site environment for internal reasons, thus the author does not include them in the project.
  - graphicType: containing a string, which is the type of the graphic that the user wants to see in the widget. Currently, the graphics that the user can utilize are: "Doughnout", "Pie Chart", "Vertical Bar", "Stacked Vertical Bar", "Horizontal Bar", "Stacked Horizontal Bar", "Line Chart", "Scatter", "Bubble", "Pyramid", "Funnel", and "Table".
  - widgetType: Inside the 4 main categories, the selection is further divided into separate types according to which categories that wanted to be visualized. Agent category can show the status of the agents, such as the number of agents that are connected, disconnected, or in quarantine. Threats category can show the status of the threats that are detected by SentinelOne, such as the number of threats that are detected, resolved, or in quarantine. Applications category show the number of outdated applications that can potentially be a threat to the client's machine, according to the latest CVEs from the NIST and MITRE. The widget categorization is divided as follows:
    - Endpoint: Agent versions, pending updates, agent by OS, console migration status, domains, encrypted applications, endpoint connected to management, connection status, endpoint health, endpoint list, endpoint summary, installers, load saved filters, local configurations, location IDs, machine types, manual actions required, network health, OS arch, pending uninstall, and scan status.
    - Threats: Analyst verdicts, confidence levels, engines, external ticket exists, failed actions, incident status, initiator, mitigated preemptively, note exists, pending actions, reboot required, threat classification, threat list, threat status, threat summary of the week, and unresolved.
    - Applications: Risk levels, and most impactful applications according to SentinelOne Vulnerability Score.
    - Miscellaneous: SentinelOne news feed, and free text.



```
timestamp: 1717414859062
widgetIds
    0 "1"
    1 "52:*(bh5d2"
    2 "&?m*imbd9s"
    3 "?:!llp/#td"
    4 ")*rcx3f034"
widgetTitle
    0 "Date Time - Threats"
    1 "Risk Levels - Applications"
    2 "Agent Versions - Endpoints"
    3 "Live Feed - Miscellaneous"
    4 "Free Text - Miscellaneous"
```

Figure 4.2: User Preference document for a specific user will contain timestamp, widgetIds, and widgetTitle



```
widgets
  0
     category: "Threats"
     graphicType: "Line"
     widgetType: "Date Time"
  1
     category: "Applications"
     graphicType: "Doughnout"
     widgetType: "Risk Levels"
  2
     category: "Endpoints"
     graphicType: "Doughnout"
```

Figure 4.3: The widgets field of the userPreference document

Chapter 5

Back-end Architecture

JavaScript

JS is a high-level, interpreted programming language that conforms to the ECMAScript specification. It is a multi-paradigm programming language, supporting OOP, imperative, and functional programming styles and is primarily used for client-side web development. It is also used for server-side development with Node.js, and for mobile app development with frameworks like React Native, Vue.js, Angular.js, and NativeScript. JS is now one of the core technologies of the web, along with HTML and CSS, and is supported by all modern web browsers. Some key features of JS include:

- Client-side Scripting: it is mainly used for client-side scripting in web browsers, allowing programmers to create dynamic content that interacts with the browser and the user. It can manipulate the content and behavior of HTML elements, respond to user actions without the need to reload the page for interacting with DOM to update page content dynamically. This enhances the UX by making web applications feel more responsive and integrated.

- Cross-Platform Compactibility: it is supported by all modern web browsers, including Chrome, Firefox, Safari, Opera, and Edge, and can be used to build cross-platform web applications that run on any device or platform, like Android or iOS.

- Server-Side Scripting: with the advent of Node.js, it has also become popular for server-side scripting. Node.js is a runtime environment that allows developers to build scalable network applications, including web servers, APIs, and real-time applications like chat servers. This has expanded the versatility and use cases of JS beyond the browsers, making it a full-stack development language.

- Dynamic Typing: it means that variables do not have predetermined types. Instead, types are determined at runtime based on the assigned values.

- Functional Programming: it supports concepts such as first-class functions, high-order functions, and closures. This allows programmers to write more concise and expressive code by treating functions as first-class citizens.

- Event-Driven Programming: it follows an event-driven programming paradigm, where actions or events trigger specific functions or code execution, makes it well-suited for building interactive UI and handling user interaction.

Node.js

is an open-source, cross-platform, server-side runtime environment built on Chrome's V8 JS engine that allows programmers to run JS code outside a web browser, enabling the development of server-side and networking applications.

TypeScript

is an open-source programming language developed and maintained by Microsoft. It is a superset of JS, meaning that any valid JS code is also valid in TS. It adds optional static typing to JS, which allows developers to annotate their code with type information, catch errors early in the development, and improve the maintainability or large codebases. Some developers prefer using TS over JS for some of its key-features:

- Static Typing: it checks the types of variables at compile-time, which enable developers to specify the types of variables, functions parameters, and return values. This introduces type safety, in contrast to JS which is dynamically typed, which means that the types of variables are determined at runtime.
- Enums: it provides support for enums, allowing developers to define a set of named constants with associated values.
- Generics: it supports generics, enabling the creation of reusable components that can work with a variety of data types.
- Decorators: it supports decorators, which are a way to add annotations and a meta-programming syntax for class declarations and members. Decorators can be used to modify classes, methods, and properties at design time, providing a powerful way to extend or modify the behaviour of classes and their members.
- Interfaces and Classes: it supports OOP concepts such as classes, interfaces, inheritance, and access modifiers, making it easier to organize and structure code.
- ES6+ Features: it supports many features introduced in ECMAScript standards beyond ES5, such as arrow functions, classes, async/await syntax, and modules.
- Backwards Compactibility: it is designed to be backwards compactible with JS, which means that developers can use it alongside JS code and integrate it into existing JS projects and environments, including, browsers, Node.js, and other JS runtime environments.

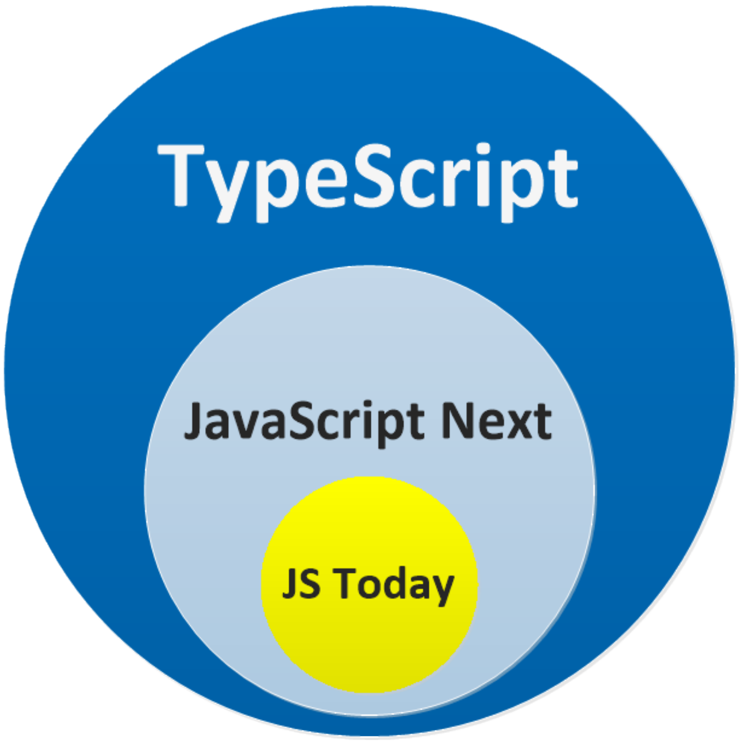


Figure 5.1: Venn diagram of TypeScript and JavaScript

As stated in the Research Report, Q-ICT wishes to utilize the 2nd generation of Firebase Cloud Functions, therefore a new Firebase project was created and stored on the cloud (*using Azure DevOps*). The author also needs to make a decision in regards on how the infrastructure of the codebase should be structured. Because Q-ICT is always critical and open to feedback, the author is given access to the old Firebase codebase (*utilizing version 1.0*), and find any potential upsides and downsides of that project repository. The author then, by an informed decision, is allowed to decide whether to structure the new codebase in the same way as the old one. The author has certainly decided to create some new adjustments to the new codebase. For example, instead of stacking all functionalities that a Firebase Cloud Function might have, the author has decided to improve the codebase especially regarding the separation of concerns. Specifically, the response from the API is modelled using Interfaces and Classes object within the Models directory, allowing for a consistent reuse across different functions. Additionally, distinct Routers and Controllers directories has been established, each with specific responsibilities. The Routers directory primarily handles external communications with the API, utilizing `Axios` (*npm, n.d.*) for handling the GET requests. The Controllers directory, contains the back-end logic of the cloud function, bridging Models and Routers and ensuring comprehensive error documentation. This structure defines the behavior of the Firebase Cloud Function version 2.0 of the QaaS app.

Moreover, there are Utilities and Middlewares directories, which are used to handle common functionalities and errors. The Utilities directory contains functions that are used across the codebase, such as logging errors, and the Middlewares directory contains functions that are used to handle the request and response of the cloud function. For example, the `logError` function in the Utilities directory is used to log errors in the console, and the `handleError` function in the Middlewares directory is used to handle errors in the response of the cloud function. This structure allows for a more organized and maintainable codebase.

5.0.1 Compliance with ESLint compiler

Additinally, the back-end project utilizes Node.js as the runtime environment, with TS as the programming language. This has caused longer compilation time as it involves additional steps like type checking and transpiling the code to JS. However, it is still believed to be the better practice as JS in nature is a loosely typed language, and it is easy to make mistakes in the code as its variables do not have a fixed type. While this sometimes can be beneficial as it makes the development faster and more flexible, it also introduces challenges, particularly in debugging and maintaining the code. To prevent this, both the author and the Company Supervisor have decided to use and adhere to TS and ESLint rules, to ensure static typing, code quality and consistency to the coding standards.

Listing 5.1: Example of Class and Interface defined in Models

```

export interface Customer {
  customerId: string;
  customerName: string;
  orgUnitType: string;
  parentId: string;
  city: string;
  stateProv: string;
  country: string;
  county: string | null;
  postalCode: string;
  contactEmail: string;
}

/**
 * Class for CustomerModel.
 */
```

```
export class CustomerModel {
  private readonly firestore: FirebaseFirestore.Firestore;
  private readonly collectionName = "customers";
  private readonly context = "CustomerModel";

  constructor() {
    this.firestore = admin.firestore();
  }

  serializeCustomerToJson(customer: Customer): string {
    return JSON.stringify(customer);
  }

  /**
   * Deserializes a JSON string to a Customer object.
   * @param {string} json The JSON string to deserialize.
   * @return {Customer} The deserialized Customer object.
   */
  deserializeJsonToCustomer(json: string): Customer {
    return JSON.parse(json) as Customer;
  }

  /**
   * Creates a new customer in Firestore.
   * @param {Customer} customer Customer to add to Firestore.
   * @return {Promise<FirebaseFirestore.DocumentReference<FirebaseFirestore.DocumentData>> | undefined}.
   */
  async createCustomer(customer: Customer):
       Promise<FirebaseFirestore.DocumentReference<FirebaseFirestore.DocumentData> | undefined> {
    try {
      return this.firestore.collection(this.collectionName).add(customer);
    } catch (ex: unknown) {
      logError(ex, this.context + ".createCustomer");
    }
    return undefined;
  }

  /**
   * Retrieves a customer by its ID from Firestore.
   * @param {string} customerId The ID of the customer to retrieve.
   * @return {Promise<Customer | undefined>} A promise that resolves with the customer object if found,
        or undefined if not found or
   an error occurs.
   */
  async getCustomer(customerId: string): Promise<Customer | undefined> {
    try {
      const doc = await this.firestore.collection(this.collectionName).doc(customerId).get();
      if (!doc.exists) {
        return undefined;
      }
      return doc.data() as Customer;
    } catch (ex: unknown) {
      logError(ex, this.context + ".getCustomer");
      return undefined;
    }
  }

  /**
   * Updates an existing customer in Firestore.
   * @param {string} customerId The ID of the customer to update.
   * @param {Partial<Customer>} customer The partial customer object containing updates.
   * @return {Promise<void>} A promise that resolves when the update is complete.
   */
  async updateCustomer(customerId: string, customer: Partial<Customer>): Promise<void> {
    try {
      await this.firestore.collection(this.collectionName).doc(customerId).update(customer);
    } catch (ex: unknown) {
      logError(ex, this.context + ".updateCustomer");
    }
  }

  /**
   * Deletes a customer from Firestore.
   * @param {string} customerId The ID of the customer to delete.
   * @return {Promise<void>} A promise that resolves when the customer is successfully deleted.
   */
  async deleteCustomer(customerId: string): Promise<void> {
    try {
      await this.firestore.collection(this.collectionName).doc(customerId).delete();
    } catch (ex: unknown) {
      logError(ex, this.context + ".deleteCustomer");
    }
  }
}
```

Listing 5.2: An example of Firebase HTTP Request onCall Cloud Function version 1.0

```
import * as functions from "firebase-functions";
import * as admin from "firebase-admin";
import { Secrets } from "../Firebase/Secrets";
import { FirebaseCall } from "../Firebase/FirebaseCall";
import axios from 'axios';

export default functions
  .region("europe-west1")
  .https.onCall(async (data, context) => {
      try {
        // context.app will be undefined if the request doesn't include a valid
        // App Check token.
        if (context.app === undefined) {
          throw new functions.https.HttpsError(
            "failed-precondition",
            "The function must be called from an App Check verified app."
          );
        }
      } catch (error) {
        console.error("An error occurred in the Firebase HTTP Request onCall Cloud Function: ", error);
        throw new functions.https.HttpsError("internal", "An error occurred while processing the
             request.");
      }
});
```

Listing 5.3: onCall Cloud Function version 2.0, where the parameters of the function is less and the way it handles the authorization is different

```
import axios from "axios";
import * as functions from "firebase-functions/v2";
import { CallableRequest } from "firebase-functions/v2/https";
import admin from "firebase-admin";

import { region, sentinelOneURL, sentinelOneApiVersion } from "../../../config";
import { logError } from "../../../Middleware/LogError";
import { getSecret } from "../../../Util/Secret";

export const getSentinelOneData = functions.https.onCall({ region: region }, async (context:
    CallableRequest<any>) => {
  try {
    // Authentication check
    if (!context.auth) {
      throw new functions.https.HttpsError("unauthenticated", "The function must be called while
          authenticated.");
    }
    const data = context.data;

    // // Checking if the request contains the necessary data
    if (!data || !data.siteId || !data.userId || !data.dataType) {
      throw new functions.https.HttpsError("invalid-argument", "The necessary requirement(s) are
          missing or invalid in your request.");
    }
  } catch(ex: unknown) {

  }
});
```

Listing 5.4: LogError functionality in the Utilities folder

```
export function logError(error: unknown, context: string): void {
  if (isAxiosError(error)) {
    // AxiosError object, log detailed request and response information
    console.error(`[${context}] Axios error occurred: ${error.message}`);
    console.error(`Response data: ${error.response?.data}`);
    console.error(`Status code: ${error.response?.status}`);
    console.error(`Headers: ${JSON.stringify(error.response?.headers, null, 2)}`);
  } else if (error instanceof Error) {
    // Standard Error object, log message and stack
    console.error(`[${context}] An error occurred: ${error.message} \n Stack: ${error.stack}`);
  } else {
    // Non-Error object, log with a generic message
    console.error(`[${context}] An unknown error occurred:`, error);
  }
}
```

Listing 5.5: An example of how that utility is used in the Cloud Function, the reason why exception type is unknown is to comply with the ESLint rules that does not recommend to use the any type to the variables

```
try {

}
catch (ex: unknown) {
  console.log(`An error occured in the getFirestoreData function: ${ex}`);
  logError(ex, "getFirestoreData");
} finally {
```

```javascript
    throw new functions.https.HttpsError("internal", "An error occurred while getting the data in Firestore.");
  }
```

Listing 5.6: Example of challenges in JavaScript because of its loosely typed nature that creates lack of type safety, type coercion, and type conversion

```javascript
function add(a, b) {
  return a + b;
}
console.log(add(5, '2')); // "52" instead of 7
```

Listing 5.7: Another example of JavaScript challenges

```
C:\Users\ChristopherSulistiyo>node
Welcome to Node.js v22.1.0.
Type ".help" for more information.
> true === 1
false
> true + true + true === 3
true
> 0 == "0"
true
> "0" ==- null
true
> -null
-0
> -0 == "0"
true
> parseInt(0.00000005)
5
> typeof(NaN);
'number'

> let a = []
undefined
> a.length == a
true
```

5.1 SentinelOne NPM packages

NPM packages are a necessary component when developing a project in JavaScript, as they are used for managing dependencies in Node.js projects. Below are the NPM packages that are used in the project:

- `axios`: Used for making HTTP requests from the Firebase Cloud Functions to the SentinelOne API.
- `firebase-admin`: Used for interacting with the Firebase services, such as Firestore and Secret Manager.
- `firebase-functions`: Used for creating Firebase Cloud Functions. Version 2.0 is used in the project. Some differences between version 1.0 and 2.0 are the way the functions are defined, the way the functions handle the request and response, and the way the functions handle the authentication. This package has many types of functions, and the ones that are used are: CallableRequest, onRequest, onSchedule, and onCall.
- `algoliasearch`: Used for integrating the Algolia search engine with the Firebase Cloud Functions. Algolia is used to make the search faster and more accurate.
- `firebase-functions-test`: Used for testing the Firebase Cloud Functions. The package provides a way to test the functions locally before deploying them to the Firebase Cloud.

5.2 Downloading PDF and HTML report files from SentinelOne to the client

At almost the end of the realization phase, based on a work item requested by the stakeholders on the SCRUM board, the author has decided to include showing the reports that have been made automatically or manually by SentinelOne or the cybersecurity actors on Q-ICT's SentinelOne environment. The process initially was quite low on the priority list (Could have on the MOSCOW), SentinelOne already has an API that will give the ASCII characters of the PDF, but calling this directly on the client itself is not possible due to CORS blocking policy. Therefore, a Cloud Function is needed as a middleman for getting the PDF file to the client.

First on the back-end, when using `axios` to call the appropriate SentinelOne URL, another header needs to be included to specify that the expected response format is PDF. Additionally, the response type also need to be set to `arraybuffer` to hand;e binary data efficiently.

Lastly, upon receiving the request, the binary data is converted into a base64-encoded string. This is necessary to ensure that the binary data can be safely transmitted and displayed within the web client. Additionally, the content of the MIME type of the response is extracted from the headers to facilitate proper handling of the data on the client side.

When the client accepts the response, it is necessary the base64-encoded string is decoded into a byte of array. Then, to facilitate the download of the PDF, a `Blob` is created from the decoded byte array. An object URL is then generated from this `Blob`, and an anchor element is programmatically created and clicked to trigger the download.

This way, a PDF or HTML file can be downloaded on the client side using any type of web browser when running the QaaS app.

```

class Data extends BaseModel {
  Data({
    required this.id,
    required this.name,
    required this.age,
  });

  factory Data.fromJson(Map<String, dynamic> json) {
    return Data(
      id: json['id'] as String,
      name: json['name'] as String,
      age: json['age'] as int,
    );
  }

  final String id;
  final String name;
  final int age;

  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'name': name,
      'age': age,
    };
  }
}

static Future<List<Data>> getData() async {
  try {
    final HttpsCallable callable =
        FirebaseFunctions.instanceFor(region: 'europe-west3').httpsCallable(
      'getData',
      options: HttpsCallableOptions(
        timeout: const Duration(seconds: 540),
      ),
    );

    final Map<String, dynamic> headers = <String, dynamic>{
      'dataType': 'abcd',
    };

    final HttpsCallableResult<dynamic> results = await callable.call(headers);
    final Map<String, dynamic> responseData =
        results.data as Map<String, dynamic>;
    final dynamic data = responseData['data'];

    if (data is List<dynamic>) {
      return data
          .map((dynamic json) =>
              Data.fromJson(json as Map<String, dynamic>))
          .toList();
    } else if (data is Map<String, dynamic>) {
      return <Data>[Data.fromJson(data)];
    }
  } on FirebaseFunctionsException catch(e) {
    if (kDebugMode) {
      print('Firebase error when fetching the data: $e');
    }
  } catch (e, stack) {
    if (kDebugMode) {
      print('Application error: $e \nStack: $stack');
    }
  }
  return List<Data>.empty();
}
```

Listing 5.8: How calling and modelling a response works in Flutter

Listing 5.9: How downloading a PDF/HTML file works on the back-end

```javascript
const response = await axios.get(urlName, {
  headers: {
    "Authorization": `ApiToken ${apiToken}`,
    "Accept": "application/pdf",
  },
  responseType: "arraybuffer",
});

const answer = {
  data: Buffer.from(response.data, "binary").toString("base64"),
  contentType: response.headers["content-type"],
};

return { data: answer };
```

```dart
static Future<void> downloadPdfReports({
  required String reportId,
}) async {
  try {
    final HttpsCallable callable =
        FirebaseFunctions.instanceFor(region: 'europe-west3').httpsCallable(
      'getReports',
      options: HttpsCallableOptions(
        timeout: const Duration(seconds: 540),
      ),
    );

    final Map<String, dynamic> headers = <String, dynamic>{
      'reportId': reportId,
    };

    final HttpsCallableResult<dynamic> results = await callable.call(headers);
    final Map<String, dynamic> responseData =
        results.data as Map<String, dynamic>;
    final String base64Data = responseData['data'] as String;

    // Decode the base64 string to bytes
    final Uint8List bytes = base64Decode(base64Data);

    // Create a Blob from the byte data
    final html.Blob blob = html.Blob([bytes], 'application/pdf');

    // Create an object URL from the Blob
    final String url = html.Url.createObjectUrlFromBlob(blob);

    // Create an anchor element and trigger a download
    html.AnchorElement(href: url)
      ..setAttribute('download', 'report.pdf')
```

```
          ..click();

        // Optionally revoke the Blob URL after some time to release resources
        Future<void>.delayed(
            const Duration(seconds: 5), () => html.Url.revokeObjectUrl(url));

        if (kDebugMode) {
          print('PDF download triggered');
        }
      } on FirebaseFunctionsException catch (ex) {
        if (kDebugMode) {
          print('Firebase error when downloading the PDF: $ex');
        }
      } catch (e, stack) {
        if (kDebugMode) {
          print('Application error: $e \nStack: $stack');
        }
      }
    }
```

Listing 5.10: How downloading a PDF/HTML file works on the front-end

Chapter 6

Front-End Architecture

The front-end architecture is a critical aspect of the QaaS App, ensuring that the UI is intuitive, responsive, user-friendly and secure. The front-end is responsible for handling user interactions, displaying data, and communicating with the back-end services. This chapter outlines the front-end architecture, including the technologies used, structure, design patterns, design principles and key components used to develop the client-side application, primarily using Flutter.

As was said before, the front-end of the QaaS App is built using Flutter, a modern UI framework that enables the development of high-performance applications for multiple platforms into a single codebase. The architecture in this project follows the MVC (Mode-View-Controller) architectural pattern to separate concerns and improve maintainability, scalability, and testability. The front-end is responsible for handling user interactions, displaying data, and communicating with the back-end services.

6.1 Flutter

Flutter is an open-source framework made by Google in 2017. It used as an UI toolkit for building natively compiled applications for mobile, web, and desktop (Windows, macOS, Linux) from a single codebase.

Widgets

Widgets in Flutter refer to the basic building blocks used to build UIs. Everything in Flutter, from buttons to layout containers to text inputs, is a widget. They are lightweight, reusable components that define the structure, appearance, and behavior of UI elements in a Flutter application. When a widget's state changes, the widget rebuilds its description, which the framework diffs against the previous description to determine the minimal changes for rendering in the UI. There are only 2 types of Widgets in Flutter:

- **Stateless Widget**: is immutable, meaning that their properties cannot change once they are initialized. They are used for static content that does not require user interaction. Examples of a stateless widget would be text labels, buttons, icons, and static images.
- **Stateful Widget**: it maintains state, allowing their properties to change over time in response to user interactions or other events. They are used for dynamic content that require updates, such as animations, form inputs, and interactive elements. Stateful widgets consist of two classes: the widget class itself and a corresponding state class that manages the widget's state. Examples of stateful widgets are: text input fields, scrollable lists, tabs, and progress indicators.

- **Inherited Widgets**: facilitate state management by allowing state to be passed down the widget tree efficiently. This is particular useful for sharing state across many child widgets.

Flutter apps are built using a hierarchy of nested widgets, forming a ”widget tree”. The root widget defines the overall app structure, and other widgets branch out to create more complex UIs.

```
// Stateless Widget
class MyTextWidget extends StatelessWidget {
      final String text;

      MyTextWidget({Key key, this.text}) : super(key: key);

      @override
      Widget build(BuildContext context) {
      return Text(text);
      }
}

// Stateful Widget
class MyCheckbox extends StatefulWidget {
      @override
      _MyCheckboxState createState() => _MyCheckboxState();
}

class _MyCheckboxState extends State<MyCheckbox> {
      bool isChecked = false;

      @override
      Widget build(BuildContext context) {
            return Checkbox(
                  value: isChecked,
                  onChanged: (bool value) {
                        setState(() {
                        isChecked = value;
                  });
            });
}
}
```

Listing 6.1: Example of a stateful and stateless widget

6.2 Architecture Components

6.2.1 State Management

Effective state management is crucial for building responsive and performant applications. The following state management solutions are considered when making this project:

- Provider: a wrapper around Inherited Widgets, providing a simple way to manage and access state throughout the application.
- Riverpod: an improvement over Provider, offering a more robust and testable state management solution.
- Bloc (Business Logic Component): a reactive state management library that uses streams to manage state changes, ensuring a clear separation between business logic and UI.

6.2.2 Routing and Navigation

Flutter provides a robust navigation system to manage routes within the application. The following approaches are used for routing:

- Pushing the stack: navigating to a new screen by pushing it onto the navigation stack. It is a traditional imperative approach for managing navigation stack and transitions between screens.
- Pop and push: navigating to a new screen by popping the current screen and pushing it onto the navigation stack. It is a declarative approach offering more control over the navigation stack, suitable for complex routing scenarios.
- Navigation with arguments: passing arguments to the next screen.

6.2.3 UI/UX Design Principles

- Responsive Design: ensure the application layout adapts to different screen sizes and orientations using Flutter's layout widgets such as `Flexible`, `Expanded`, and `MediaQuery`.
- Material Design: follow Google's Material Design guidelines for consistent and intuitive user interfaces and experience. Use Flutter's built-in Material components.
- Accessibility: ensure the application is accessible to all users and meet accessibility standards. Use Flutter's accessibility features such as `Semantics` and `FocusNode`. Features included are screen reader support, high-contrast themes, and accessible navigation to ensure the app is usable by all users.

6.2.4 Security

- HTTPS: ensure all data communication between the client and server is encrypted using HTTTPS whether it is at transit or at rest. This will be mainly used when talking to the Firebase Cloud Functions.
- CORS: use Cross-Origin Resource Sharing to enable cross-origin resource sharing. This is important when the app is deployed on a web server.
- Input Validation: validate all user inputs on the client side to prevent common security vulnerabilities such as cross-site scripting (XSS), and data tampering and to ensure data integrity.

6.2.5 Performance Optimization

- Lazy Loading: implement lazy loading of data in Flutter apps by using the `FutureBuilder` widget. This will ensure that data is loaded only when needed, improving app performance and reducing resource consumption. It improves initial load times and reduce memory usage when loading data-heavy components. A spinning MKBiT logo has been chosen as the loading indicator.
- Efficient Rendering: use Flutter's built-in performance profiling tools to identify and optimize rendering bottlenecks.

- Asynchronous Operations: leverage Dart's asynchronous single-threaded programming features to perform network requests and other long-running operations without blocking the UI.

The front-end architecture of the Qaas App is designed to provide a responsive, secure, and maintainable UI. By leveraging Flutter's powerful UI framework and following the MVC architectural patterns, the architecture ensures a clear separation of concerns, efficient state management, and an intuitive UI. This foundation will support the seamless integration of SentinelOne EDR and provide clients with transparent view of their device security information, and providing a clear pathway in the development of future SentinelOne modules that are yet to be integrated.

Chapter 7

Test Strategy

This test strategy outlines the approach for system testing the integration of SentinelOne EDR with the QaaS App. The goal is to ensure that the system functions as expected, meets the specified requirements, and provides a reliable and secure experience for users.

7.1 Test Objectives

The primary objectives of system testing are to:

- Verify that the integration with SentinelOne API functions correctly.
- Ensure that data is accurately fetched, displayed, and updated.
- Validate user interactions and interface elements.
- Confirm that security measures are effective.

7.2 Test Scope

System tests will cover the following components:

1. Device Health and Status Dashboard.
2. Audit Trail and Threat Information.
3. Client Management.
4. Integration with SentinelOne API.
5. Database interactions with Firebase Firestore.
6. Front-end components developed using Flutter.

7.3 Test Types

7.3.1 Functional Testing

Functional tests will verify that each feature works according to the specified requirements.

- Dashboard: verify the display of device health and status information.

- Audit Trail: check the accuracy and completeness of the threat audit trail.
- Client Management: validate the addition, update, and removal of clients and devices.
- API Integration: ensure data is correctly fetched and processed from SentinelOne API.

7.3.2 Performance Testing

Performance tests will assess the system's responsiveness and stabilitty under load.

- Load Testing: test the system's responsiveness under various load conditions by simulating multiple users accessing the system simultaneously to evaluate performance especially in the database and cloud functions.
- Stress Testing: identify the system's breaking point by gradually increasing the load beyond normal usage levels.
- Scalability Testing: evaluate the system's ability to scale horizontally by adding more clients and devices to the system and ensuring the system can scale horizontally to handle increased load.

7.3.3 Usability Testing

Usability test will assess the UX and interface design of the system.

- UI: ensure the UI is intuitive and easy to navigate.
- Accessibility: test the system's accessibility features such as screen reader support and high-contrast themes. Verify that the application meets accessibility standards.
- Responsiveness: ensure the application functions correctly on various devices and screen sizes.

7.4 Test Environment

The test environment will mimic the production environment closely as possible, including:

- Backend: Firebase Cloud Functions and Firebase Firestore.
- Frontend: Flutter application running mostly on web.
- Integration: SentinelOne API.
- User Devices: various desktop browsers, mobile smartphone devices, and tabloid devices.

7.5 Test Schedule

- Planning and Preparation: 1 week.
- Development of Test Cases: 1 week.
- Execution of Tests: 2 weeks.
- Bug Fixing and Retesting: 3-4 weeks.
- Final Verification and Reporting: 1 week.

Please also note that during the development of this project, the author only had access to the test environment of both the server and the client. This means that the author only has limited access to the data that is supposed to be shown and even less permission access to the SentinelOne API like POST, PUT, PATCH, and DELETE.

Chapter 8

Security Considerations and Author's Remark

8.1 Structure of the front-end codebase

There are some issues that the author has noticed with the way the QaaS app code was structured. For example, when assigning modules to a user, sometimes the user can see the modules that are not assigned to them. This may due to the fact that the code is not structured properly, and that they were some bugs left unchecked during the first development of the QaaS app. According to the Company Supervisor, the QaaS app itself is still young, so there are many rooms for improvement, especially regarding the structure of the codebase and the lack of utilization of software design patterns. The author that there should be a separate project, intended directly on refactoring the QaaS app, to make it more robust and scalable in the future.

Furthermore, there are some inconsistencies in the naming of the variables, especially regarding user tags and user roles. They are all doing the same thing, but the naming is different. This due the fact that this project was initially a continuation from an unfinished project, and the current developer did not bother to change the naming of the variables. This inconsistency can lead to confusion for the future developers, and it is recommended to change the naming of the variables to be more consistent.

8.2 Translation

The target audience of this project is Q-ICT's customers, in which all of them are a Dutch company. Therefore, the author and the Company Supervisor have tried to make the SentinelOne page in the QaaS to be in Dutch with the text buttons and the table headers, and the general UI widgets. However, as SentinelOne itself is an American company, the data that is fetched from the API is in English. The author and the Company Supervisor have not yet explored the idea of using an external package in the back-end that can connect to a translation party, like Google Translate or Microsoft Translator API, to translate the data. This feature, as the stakeholders have mentioned, would be a nice addition to have, as most of the clients do not know any technical knowledge of Cybersecurity, and having the data in their native language would make it easier for them to better understand the data.

8.3 Future modelling

The modelling in both the back-end and front-end are done in a way that is suitable for JSON response from the API. However, the author has not yet explored the idea of making the modelling to be more future-proof, by making the modelling to be more dynamic, by adding other schemas such as XML, YAML, or CSV. This way, the QaaS app can be more flexible in the future, and in case SentinelOne wants to also expand their API server, for example by adding a SOAP server that returns XML response.