Cybersecure REST API for Manufacturing Execution Systems

Otto Lönnbäck

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 22.12.2022

Supervisor

Prof. Janne Lindqvist

Advisor

M.Sc. (Tech.) Dimitar Boyadzhiev

Aalto University
School of Science

Copyright © 2022 Otto Lönnbäck

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Otto Lönnbäck

Title Cybersecure REST API for Manufacturing Execution Systems

Degree programme Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Prof. Janne Lindqvist

Advisor M.Sc. (Tech.) Dimitar Boyadzhiev

Date 22.12.2022 **Number of pages** 56+1 **Language** English

Abstract

Legacy systems are still in widespread use, especially in corporate settings and factories. They often use proprietary communication protocols, making it harder to integrate them with new software and third-party applications. Implementing REST APIs for such systems can improve their interoperability and ease of use. When implementing new APIs for legacy systems security is of particular concern.

This thesis investigates the security considerations of building a REST API for a legacy system. It does this in the form of a case-study of the development of a secure REST API backend for a Manufacturing Execution System. The thesis investigates how the REST API can be implemented in a secure manner, and which security aspects have to be considered. It also investigates special security considerations for legacy systems.

The legacy system in question relies heavily on logic running in the database, which led to a focus on an API that directly interfaces with the database. This choice in turn put a focus on proper validation and authorization procedures. For interoperability, some security relevant systems, such as authentication, were made compatible with the legacy system.

For implementing the API a model-driven development approach is used, where the meta-model is based on the OpenAPI Specification (OAS). The meta-model is highly domain-specific in order to incorporate aspects such as authorization and validation. It is based around the concept of database tables and rows being modeled as REST resources. In addition to this, traditional non-model-driven solutions are also used for some security related functionality.

The resulting implementation was found to be secure, but there were cases where more secure options than those implemented were available. Particularly, compatibility with the legacy systems required some compromises and special considerations. The use of model-driven development was found to improve the security of the API backend.

Keywords API security, REST API, model-driven development, model-driven security, OpenAPI, legacy systems

Aalto University
School of Science

Aalto-universitetet, PB 11000, 00076 AALTO
www.aalto.fi
Sammandrag av diplomarbetet

Författare Otto Lönnbäck	
Titel Cybersäkert REST API för produktionsstyrningssystem	
Utbildningsprogram Computer, Communication and Information Sciences	
Huvudämne Computer Science	**Huvudämnets kod** SCI3042
Övervakare Prof. Janne Lindqvist	
Handledare DI Dimitar Boyadzhiev	
Datum 22.12.2022 **Sidantal** 56+1	**Språk** Engelska

Sammandrag

Äldre system används fortfarande i stor utsträckning, speciellt inom företag och i fabriker. De använder ofta proprietära kommunikationsprotokoll, vilket gör det svårare att integrera dem med ny mjukvara och program från tredje parter. Att implementera ett REST API för sådana system kan förbättra deras interoperabilitet och lättanvändlighet. När man implementerar nya gränssnitt för befintliga system är speciellt säkerheten en viktig angelägenhet.

Det här diplomarbetet undersöker de säkerhetsangelägenheter som bör tas i beaktande vid utvecklingen av ett REST API för ett befintligt system. Detta uppnås genom en fallanalys om utvecklingen av ett säkert REST API för ett produktionsstyrningssystem. Arbetet undersöker hur ett REST API kan implementeras på ett säkert sätt, samt vilka säkerhetsaspekter som behöver tas i beaktan. Det undersöker också vilka specialbeaktanden rörande säkerheten som behövs för ett befintligt system.

Det befintliga systemet i fråga beror kraftigt på logik som körs i databasen, vilket ledde till ett fokus på ett gränssnitt som direkt kommunicerar med databasen. Det här valet ledde i förlängningen till ett fokus på korrekta validerings- och aktoriseringsprocedurer. För att bibehålla kompitabiliteten med det befintliga systemet var man tvungen att återanvända vissa existerande system, till exempel för autentikering.

För att implementera API:t används modelldriven utveckling, där metamodellen är baserad på OpenAPI Specification (OAS). Metamodellen är väldigt domänspecifik för att inkorporera aspekter såsom validering och auktorisering. Den är baserad kring konceptet att databasens tabeller och rader modelleras som REST-resurser. Utöver detta används även traditionella ickemodelldrivna lösningar för att implementera viss säkerhetsrelaterad funktionalitet.

Den resulterande implementeringen befanns i allmänhet vara säker, men det fanns vissa fall där mer säkra alternativ än de implementerade fanns tillgängliga. I synnerhet kompitabiliteten med det befintliga systemet krävde vissa kompromisser och speciallösningar. Användningen av modelldriven utveckling befanns förbättra systemets säkerhet.

Nyckelord API-säkerhet, REST API, modelldriven utveckling, modelldriven säkerhet, OpenAPI, befintliga system

Preface

I want to thank my supervisor Professor Janne Lindqvist and my advisor Dimitar Boyadzhiev for their guidance when writing the thesis. I would also like to thank ABB for providing me with a thesis position.

Helsinki, 22.12.2022

Otto Lönnbäck

Contents

Abstract 3

Abstract (in Swedish) 4

Preface 5

Contents 6

Symbols and abbreviations 8

1 Introduction 9

2 Background 11
- 2.1 HTTP . . . 11
- 2.2 REST . . . 12
- 2.3 OpenAPI . . . 13
- 2.4 Zero trust . . . 14
- 2.5 OWASP API Top Ten . . . 14
- 2.6 Browser vulnerabilities . . . 16
- 2.7 HTTPS and TLS . . . 17
- 2.8 Security relevant headers and browser security features . . . 18
- 2.9 Authentication . . . 21
- 2.10 Authorization . . . 21
- 2.11 Model Driven Development . . . 22
- 2.12 Model Driven Security . . . 22
- 2.13 Using MDD to build a REST API . . . 23
- 2.14 OpenAPI as the base for a meta-model . . . 23
- 2.15 Oracle Database . . . 23

3 Implementation 25
- 3.1 Legacy ABB MES backend and frontend . . . 25
- 3.2 Requirements for the new backend . . . 26
- 3.3 Technology choices . . . 29
- 3.4 Motivation for MDD . . . 31
- 3.5 Domain specific meta-model based on OpenAPI . . . 31
- 3.6 Functional extensions . . . 32
- 3.7 Request data validation . . . 35
- 3.8 Authorization . . . 36
- 3.9 Mapping between OracleDB and OpenAPI data types . . . 37
- 3.10 Deriving the OpenAPI model from existing database structures . . . 38
- 3.11 Code generation and implementation . . . 39
- 3.12 HTTPS . . . 40
- 3.13 Authentication . . . 40
- 3.14 Logging . . . 41

3.15 Security headers 41

4 Results **43**
4.1 Evaluation of the use of MDD 43
4.2 Security analysis based on OWASP Top Ten 44
4.3 HTTPS and TLS 46
4.4 Browser vulnerabilities 46
4.5 Special considerations for legacy systems 48
4.6 Remaining security issues 48

5 Summary **50**

References **52**

Symbols and abbreviations

Abbreviations

ABAC	attribute based access control
API	application programming interface
CORS	cross origin resource sharing
CRUD	create, read, update, delete
DB	database
HTTP	hyper text transfer protocol
HTTPS	hyper text transfer protocol secure
REST	representational state transfer
ReBAC	relationship based access control
RBAC	role-based access control

1 Introduction

Legacy systems are still in widespread use, especially in corporate settings and factories. There is an ever increasing need to allow outside interaction with these systems, which is hampered by their reliance on proprietary communication protocols. It would be preferable to allow secure access to the system through a well documented, standard type of API.

For a MES system, interacting with other software components is important. It is only one layer of software typically used in a factory, and sits between the Enterprise Resource Planning (ERP) and the Supervisory Control And Data Acquisition (SCADA) layers. The MES must be capable of communicating with both of these systems, and vendor lock-in is undesirable for customers of MES systems. One type of API that fits this type of system is a HTTP based REST API documented using OpenAPI.

In this thesis, we develop a REST API backend for the *ABB Manufacturing Execution System for Pulp and Paper* (ABB MES), a software system intended for use in pulp and paper mills. We focus especially on the security related aspects of the REST API backend. Implementing the backend required some bespoke solutions to properly integrate with the legacy system.

To implement the large number of endpoints required we turn to model driven development (MDD) in order to model the relationship between the REST API and the existing ABB MES software, particularly the underlying database. In this process we exploit information on the legacy system that we automatically extracted from the system, as well as expertise of the maintainers of the system. We mainly provide a framework for performing CRUD operations targeted at database tables, but allow other operations to be performed by executing existing code.

We base our meta-model on the OpenAPI specification, a widely used format for describing REST APIs. We also include security related aspects such as input validation and authorization in our meta-model. The proposed meta-model is detailed enough to use code generation to produce the controllers for the endpoints, as well as authorization and validation middleware.

We present a method to automatically generate a basis for the model based on the existing database constructs.We argue that the approach of using MDD and partial generation of the model reduces the risk of developer errors, and thus increases the security of the REST API backend.

In addition to the modeled aspects of the system, other security related aspects are also evaluated. This includes authentication, use of the security features of the HTTP protocol and testing of both code generation and the resulting REST API backend.

The aim of the thesis is to demonstrate how cyber security was taken into account in the development process of the backend and evaluate the choices that were made. In particular, we are interested in approaches that could be useful when developing REST APIs for other legacy systems.

The rest of the thesis is structured as follows. The second section covers background material, including previous research and certain technical background infor-

mation. The third section discusses the implementation of the backend, focusing on security aspects. The fourth section presents results of the research and discusses the success of the security related features of the backend. The fifth and final section summarises the thesis.

2 Background

This section presents relevant background material for the thesis and previous research. It includes presentations of relevant concepts such as HTTP, REST and OpenAPI. It also presents the OWASP API Top Ten list used as a framework for analysing the security of the REST API. It then Goes through relevant research for the thesis. Finally, it offers a quick introduction to Oracle DB.

2.1 HTTP

Hyper Text Transfer Protocol (HTTP) is the protocol used to make requests on the World Wide Web. [1] HTTP traffic can be encrypted using TLS, referred to as HTTPS. [2] A HTTP request is initiated by the client and has a method, URL, headers and an optional body containing a representation. The URL can be split into a path identifying a resource and a set of query parameters. The most common methods are listed below:

- **GET** Fetches a representation of a resource. The GET method should be idempotent.
- **POST** Requests a resource to process the enclosed representation. One specific use of the method is to create a new resource.
- **PUT** Replaces a resource with the enclosed representation.
- **PATCH** Requests a set of changes specified by the enclosed representation be applied to a resource. [3]
- **DELETE** Removes the association between a resource and its current representation(s), and may delete the representation(s).

A HTTP response has a status code, headers and an optional body containing a representation. The most common status codes are listed below:

- **1XX** Informational responses.
- **2XX** Succcessful responses. Of particular note are 200 OK, 201 Created and 204 No Content.
- **3XX** Redirection responses.
- **4XX** Client error responses. Of particular note are 400 Bad Request, 401 Unauthorized (actually unauthenticated), 403 Forbidden (unauthorized in the traditional sense), 404 Not Found and 412 Precondition Failed.
- **5XX** Server error responses. Of particular note is 500 Internal Server Error.

HTTP headers contain metadata such as content media type, caching behavior, and authentication. There are also numerous security related headers that are further discussed in section 2.8.

2.2 REST

REpresentational State Transfer (REST) is an architectural style designed for the web. REST was first defined by Roy Fielding in his PhD dissertation. REST builds on top of HTTP and provides seven architectural style constraints: client-server, stateless, cache, uniform interface, layered system and code-on-demand.[4]

The client-server architectural style constraint separates client concerns from server concerns. This leads to improved portability of the client and a simpler server which improves scalability. Finally it also allows components to evolve independently of each other.[4]

Statelessness forbids session state to be stored on the server. Thus each request must contain all information necessary to fulfill the request. Session state should be stored entirely on the client. This constraint is frequently broken in order to provide an authenticated session.[4]

Cacheability allows network efficiency to be improved, but can cause clients to use stale data if implemented incorrectly. Cacheable resources may be stored locally instead of being fetched form the server on subsequent requests. Responses must be explicitly labeled as either cacheable or non-cacheable.[4]

The uniform interface constraint states that the components of the architecture must communicate over a uniform interface. Thus implementations are decoupled from the services they provide. This constraint simplifies the overall architecture, but the trade-off is a loss in efficiency as information is transferred in a standardized form that may not be optimal for the use-case. REST defines four interface constraints for the uniform interface: "identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state."[4]

In REST a resource is identified by an URI. A resource can be any information that can be named, including static files and temporal services. One resource may have multiple representations that can be chosen from, for example an API may provide both a JSON and XML representation. The representations may change over time, but the semantics of the mapping must remain the same. Thus the functionality of any API endpoint may not change. Resources are always manipulated through representations, which can be included both in requests and responses.[4]

Self-descriptive messages contain all information needed to understand the content. This includes metadata such as media type and cacheability information. REST also stipulates that hypermedia should be used as the engine of application state. Thus the representations should include links to any related resources, so that the operation of the client can be envisioned as a set of state transitions through links. This constraint is commonly violated in APIs, as they often provide data intended for programmatic usage and not for direct human consumption. Links are inflexible and hard to understand programmatically, instead APIs often rely on documentation provided to their consumers.[4]

The sixth architectural style constraint of REST is a layered system. "The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond

the immediate layer with which they are interacting."[4] This reduces overall system complexity, allows legacy services to be encapsulated and allows for technology such as load balancing.[4]

The final architectural style constraint is code-on-demand. It allows the client to be simplified by downloading code for certain functionality on-demand. This is only an optional constraint of the REST architectural style. Code-on-demand is not desirable in a third-party API as it requires execution of untrusted code on the client.[4]

APIs that claim to follow the REST architectural style often ignore certain constraints. These constraints are statelessness, hypermedia as the engine of application state and the optional constraint code-on-demand. It may thus be concluded that these constraints are of lesser importance when developing a real world REST API.

In [5], the author claims that pure REST as an architectural style is not secure. They base this claim on the fact that the specification of REST does not include any security considerations. In particular, the constraints of statelessness, cacheability and code-on-demand are criticized for impacting security. It can thus be concluded that some deviations from the constraints are required to build a secure REST API.

2.3 OpenAPI

"The *OpenAPI Specification* (OAS) defines a standard, programming language-agnostic interface description for HTTP APIs"[6]. OpenAPI uses a JSON or YAML formatted description of a HTML API, that can be used by both humans and computers to understand the API without additional documentation. OpenAPI definitions can be used to generate documentation, but also for (client and server) code generation and testing purposes.[6]

OAS specifies endpoints using path items. Each path item has a set of (path) parameters and operations (HTTP methods). The operations have a unique operation id, (query string) parameters, an optional request body and a set of possible responses for each possible HTTP response code. The responses contain headers, links to other operations that can be reached through the response, and possibly multiple content options associating a media type with a schema. Parameter and request body objects also have schemas. Schemas are specified using a modified (as of OpenAPI v3.0) version of JSON Schema.[6]

JSON Schema provides a specification for how JSON data should be formatted. OAS uses it to provide a schema for both request and response data. The schema can specify required values of objects, contents and length of arrays, as well as the data type and format (including URL, email and date) of individual values. Other requirements that can be encoded in the schema include regular expression conformance, minimum and maximum length (strings), minimum and maximum values (numbers) and enumerations.[6]

In OpenAPI definitions security schemes can be provided both globally (applies to all operations) and per operation. The security scheme defines how the user must be authenticated to access the operation (for example a HTTP bearer token). OAS can also be extended with additional properties. These extended properties should

be prefixed by `x-`.[6]

As OpenAPI uses JSON schema to describe parameters, and both request and response bodies, it can be used for validation of request data. In [7], the authors present a formalization of JSON schema, and prove that it can be used to efficiently validate JSON data.

2.4 Zero trust

Enterprises have traditionally employed perimeter-based network security to protect their infrastructure. This has been shown to be insufficient as an attacker who breaches the perimeter is free to move laterally through the network. This has lead to the development of zero trust as a new model for cybersecurity. Zero trust assumes that an attacker is present in the system and requires authentication and authorization of every access request. This also includes connections between services in the network, such as between an application and a database. [8]

In zero trust a central controller is responsible for authentication and enforcement of access policies. Access policies should be defined according to the principle of least privilege, and should apply to all resources in the network. The network should be segmented into smaller parts, to minimise risk of movement. All communication between components should also use mTLS. [9]

2.5 OWASP API Top Ten

There exists a large number of possible vulnerabilities in REST APIs. To limit the scope of this thesis the *OWASP API Top Ten* list of ten critical web API vulnerabilities is used. The list is released by the Open Web Application Security Project, which aims to improve web app security and provides documentation and tools relating to web security. The most recent version of OWASP API Top Ten was released in 2019, and was the first list released specifically for API vulnerabilities. [10]

The list is based on two sources: publicly available data on API security incidents and the assessment of security practitioners with penetration testing experience. The vulnerabilities from both of these categories were rated by risk based on exploitability, prevalence, detectability and technical impact by the same group of security practitioners and a consensus of the ten most critical vunerabilities was formed. This list was then reviewed by another group of security experts. Unlike the standard OWASP Top Ten (for web application security), no public call for data was issued. This was due to the difficulty of obtaining data for a first version. [10]

The lists of vulnerabilities released by OWASP have been used as a framework in other security research, particularly the more established OWASP Top Ten list. There also exists some research using the OWASP API Top Ten list, but it is more sparse. [11] [12] [13] The usage of the list in this thesis was motivated by the existing interest in the vulnerabilities included in these lists.

The vulnerabilities included in OWASP API Top Ten are in order: broken object level authorization, broken user authentication, excessive data exposure, lack of

resources & rate limiting, broken function level authorization, mass assignment, security misconfiguration, injection, improper assets management and insufficient logging & monitoring. These categories of vulnerabilities are discussed more thoroughly below, together with related research. [10]

Broken Object level authorization is the first vulnerability in the list. It relates to authorization to single objects. An object may for example be a row in a relational database. This issue is especially prevalent in APIs as they then to expose endpoints that handle object identifiers. OWASP proposes that object level access control should be considered for all endpoints. Access control models are discussed in more detail in 2.10. [10]

Broken user authentication is the second vulnerability. It relates to incorrectly implemented authentication mechanisms. Broken user authentication may allow attackers to gain access to authentication tokens or exploit flaws allowing them to impersonate other users. In particular any authentication endpoints must protect against brute force guessing of credentials and credentials-stuffing. Authentication is further discussed in 2.9. [10]

The third vulnerability, *Excessive data exposure*, relates to unnecessary exposure of sensitive information through endpoints. This is often caused by developers striving to implement generic solutions, and relying on clients to filter the exposed data. To mitigate this issue, the developer must thoroughly consider which data should be exposed through the API. [10]

Lack of resources & rate limiting is the fourth vulnerability, and concerns a lack of limiting the size of resources in both requests and responses, as well as the lack of rate limiting the number of requests. Proper limits should be included on both the size of resources and their number. This includes the number of objects in responses to client queries. Rate limiting should be imposed both on user and IP level to limit fraudulent access to the API. This vulnerability can lead to Denial of Service (DoS) attacks, either by an attacker making large numbers of requests or sending oversized payloads. [10]

Broken function level authorization is the fifth vulnerability on the list. It relates to authorizing users to access certain endpoints. Finding vulnerable endpoints in APIs is easy as they are often structured. Especially administrative functions are key targets for exploitation. [10]

The sixth vulnerability on the list is *Mass assignment*. API endpoints that do not properly validate and filter client supplied data before storing it in a database, may unintentionally allow the client to assign data to multiple objects at once, or to properties that should not be accessible for modification. Protection against this vulnerability is provided by properly filtering out any properties that should not be assignable from user input. [10]

Security misconfiguration is the seventh item on the list. Security misconfiguration can be caused by insecure defaults, incomplete configuration, misconfigured HTTP headers and too permissive Cross-Origin Resource Sharing policies. Security misconfiguration can occur at any level of the application stack. Security misconfiguration also includes the use of vulnerable or outdated components. [10]

Injection is the eight item. Injection attacks inject hostile code in a request, and

exploits vulnerabilities in the target server to make it execute said code. Injection attacks may target a database (SQL injection, NoSQL injection etc.), a website (Cross site scripting), or any other context where the server generates commands or code based on user input. The main protection against injection attacks is user input validation and sanitation. [10]

One common class of injection attack is *SQL injection*. This type of attack targets SQL databases that are commonly used for data storage. The purpose of the attack is to leak sensitive data, or alter the contents of the database. Using prepared statements and sanitizing user input are the main protections against SQL injection. [14]

The ninth vulnerability is *Improper assets management*. APIs tend to expose a large number of endpoints, and multiple versions may be deployed simultaneously. For this reason, it is important to maintain up to date documentation of the system. Old versions of endpoints may also contain vulnerabilities that have been patched in more recent versions of the API. [10]

Insufficient logging & monitoring refers to insufficient logging of requests and user activity, as well as monitoring of suspicious activity. On the other hand logs should not include sensitive information. Any user supplied data must also be neutralized before logging it. [10] Events that should be logged include authentication, authorization and access. It is important to log both successful and failed cases [16]. Logs can be used to automatically monitor for certain attacks and vulnerabilities [15].

2.6 Browser vulnerabilities

In addition to the API related vulnerabilities presented above, a client using an API can be the target of a number of browser vulnerabilities. There exists certain mitigations that can be implemented against such exploits at the API end.Therefore, we provide a quick overview of some of the most prevalent browser specific vulnerabilities.

Cross-site request forgery (CSRF) attacks perform cross-site requests from the attacking site in a web browser. The purpose of CSRF attacks is often to perform actions as an authenticated user. [17] This includes fetching sensitive data, posting forms and performing other actions. CSRF can be performed through script based fetches (e.g. the *Fetch API*), unless CORS is used, as well as through no-CORS requests such as form submission.

Cross-site script inclusion (XSSI) attacks are a variant of CSRF attacks. Unlike ordinary CSRF attacks, they rely on including scripts from the victim website in the attacking website. By doing this they try to leak sensitive data, for example through global variables. JSONP is particularly vulnerable to this type of attack, but any data that can be interpreted as Javascript, including JSON arrays is potentially vulnerable. [18]

Cross-site scripting (XSS) attacks inject hostile code into a website. XSS requires the website to include the attackers code. This can for example be performed if the website includes strings from an API directly in the HTML, and a string contains a `<script>` tag. XSS attacks allow the attacker to execute arbitrary code in the context of a website. It can be used to steal credentials or perform any requests from

the compromised website. The style of XSS attack that is most relevant for APIs is persistent XSS attacks where a script is stored in the database. [19]

2.7 HTTPS and TLS

The HTTP protocol has an encrypted variant, *Hyper Text Transport Protocol Secure* (HTTPS). HTTPS tunnels the HTTP traffic through a *Transport Layer Security* (TLS) connection. [2] TLS provides guarantees of authenticity (only for the server unless using mTLS), integrity and confidentiality. TLS includes a number of cipher suites. A cipher suite is a combination of cryptographic algorithms, generally a key exchange algorithm, a bulk encryption algorithm and a message authentication code. HTTPS can protect against MITM and impersonation attacks, provided that the used cipher suites are secure. TLS generates a set of long-term public and private keys on handshake, and then uses these to generate short-term symmetric keys used to encrypt the communication. [20]

Older versions of TLS (and its predecessor SSL) are vulnerable to a number of attacks. Currently, only versions 1.2 and 1.3 are considered safe. It is important not to allow connections using older protocol versions than this. TLS 1.3 removed support for many insecure cipher suites and obsolete features that have been proven insecure such as compression and renegotiation support. Only cipher suites that provide forward secrecy (previous communication cannot be decrypted if the private key of the certificate is compromised) are included in TLS 1.3. [22]

TLS relies on X.509 certificates sent by the server to verify the identity of the server. These certificates include information such as intended usage, target domains, a public key, issuer and validity period. The certificate is signed by the issuing *Certificate Authority* (CA) to prove its authenticity. There must exists a chain of trust enforced by signing successive certificates all the way up to a root certificate that is inherently trusted by the client. [21]

The ability of the ciphers to verify the identity of the server relies on a number of assumptions. Firstly, that no CA in the chain of trust is compromised or otherwise issues fraudulent certificates. Secondly, that no one except the CAs has access to the private key used to sign certificates further down the chain. Finally, that the client's list of trusted root certificates is not tampered with to include untrusted certificates. If any of these assumptions is violated, the servers identity can no longer be trusted. [23]

Certificate revocation is the process of revoking fraudulent or compromised certificates. There are three methods in use to revoke certificates, *Certificate Revocation Lists* (CRL)[21], *Online Certificate Status Protocol* (OCSP)[24] and *OCSP Stapling*[25]. CRLs are published by CAs and contain all certificates issued by said CA that have been revoked before their expiration date[21]. CRLs contain a large amount of data and are not directly used by clients to verify certificate validity. Instead, TLS clients such as web browsers include lists of revoked certificates in their releases. OCSP is a protocol that allows clients to verify the revocation status of certificates by querying an OCSP Responder (a server run by the issuing CA). OCSP Stapling includes the OCSP response of the used certificate in the TLS handshake.

The response is timestamped and signed by the issuing CA with a limited time of validity. A certificate can mandate OCSP Stapling by including the MustStaple extension. When using OCSP stapling it is the server, and not all clients that send OCSP requests to the CA, reducing traffic.

Certificate Transparency (CT) provides a system of public logs of valid certificates. The CT logs are append only and each issued certificate with CT support must include a reference to a log in which it is included in the form of a *Signed Certificate Timestamp*. These logs may be queried by a CT monitor that compares the issued certificates against the correct ones used by the domain owner. This allow a domain owner to find any fraudulent certificates issued for their domains, and revoke them. Clients that are CT-enabled do not accept certificates that are not included in a trusted CT log. CT is not a protection against compromised certificates as these certificates are also always included in the CT logs. Instead, it is a protection against malicious or compromised CAs, that either issue certificates without CT, or issue certificates with CT to the wrong party. In the first case, all CT-enabled clients will reject the certificate. In the other case, clients will still accept the certificate, but the legitimate owner is made aware of the fraudulent certificate, and can request its revocation. [26]

2.8 Security relevant headers and browser security features

There are a number of security features specific to browsers that can be used to protect this class of clients. The features are of lesser importance for other clients, as they generally assume that the requests originate from a browser which adheres to certain rules. Research suggests that security headers are not currently used to the full extent possible. [27]

Security in web browsers is built around the same-origin policy (SOP). According to this policy, only requests to the same origin are allowed by default. Origins are defined as the combination of protocol, domain and port. There are a number of cases where the SOP does not apply. This includes page content such as the `<img>`, `<video>` and `<script>` tags. These tags can thus perform arbitrary GET requests, the responses to which the browser tries to interpret as the expected resource. [28]

In [29] concludes that the SOP contains numerous weaknesses and argues that the SOP should preferably be replaced by a new security model. This is due to the numerous security flaws in the model that have been found over the years.

Cookies are a feature of the HTTP protocol that allows a server to store a small amount of data in the client, that will then be sent back on each subsequent request. Cookies are domain specific. Cookies can be either *session* cookies or *permanent* cookies. Permanent cookies have either an expiry date or a maximum age. Session cookies expire at the end of a session when the browser is closed. However, some browsers include session restoration functionality that restores session cookies. Cookies can be declare as HTTP only and secure. HTTP only cookies are not accessible to client scripts and only sent in requests to the server. Secure cookies are only sent over the HTTPS protocol. [30]

The scope of a cookie can be restricted to specific domains and paths. A restricted

cookie is not sent with requests with domains or paths that are not included in the allowed domains or paths. It should be noted that setting the domain of a cookie also includes the cookie in requests to any subdomain of the specified domain, while the default behavior when not specifying a domain is to exclude the cookie from requests to subdomains. [30]

Cookies can have different behavior for cross-site requests, defined by their *SameSite* attribute. Note that sites in the context of cookies include only protocol and domain, not port. A SameSite attribute of *Strict* restricts the cookie to requests originating from the same site as the request target. A value of *Lax* otherwise functions as Strict, but includes the cookie on navigation from another site. The final option, *None*, includes the cookie in all cross-site requests. According to the current specification the default behavior if SameSite is not set should be Lax. However, this was recently changed and previously the default was None. Additionally, the protocol was not previously taken into consideration when determining if a request was same-site. [30]

HTTP provides a number of security related headers, some of which are intended to secure websites, and others that are intended to secure API endpoints. The security headers only provide a security benefit for requests originating from web browsers, where the browser is assumed to be trusted and the Javascript on a website untrusted. They do not provide any security benefit for requests from other clients as the headers sent by the client cannot be trusted and the client cannot be assumed to respect any headers sent by the server.

The *Cross Origin Resource Sharing* (CORS) set of headers allow cross-origin sharing of responses for script-initiated requests, which normally are forbidden by the same-origin policy[31]. Without CORS, it is not possible to access third party APIs from browsers.

Some requests are required to be preflighted. This involves an additional OPTIONS request to the resource before the main request. The server approves or disapproves the CORS request by responding to the preflight request with an appropriate status code. The main request is blocked if the preflight request was disapproved, or if the server does not implement CORS. [31]

Any request that could be performed as a simple form submission does not require a preflighted request, as form submission predate script-initiated requests and the CORS specification[31]. These requests are GET, HEAD and POST requests that do not programmatically sets headers other than *Accept*, *Accept-Language*, *Content-Language*, *Content-Type*, and *Range*. Additionally, the only values allowed for *Content-Type* are *application/x-www-form-urlencoded*, *multipart/form-data* and *text/plain*. These simple requests can still opt-in to using CORS by returning appropriate response headers[31].

All CORS requests, including preflight requests must include the *Origin* header, however also non-CORS requests may include this header. Additionally, preflight requests include *Access-Control-Request-Method* indicating the method of the main request, and *Access-Control-Request-Headers* indicating which headers the main request will use. [31]

The responses to all CORS-requests, including preflight requests, can include

the *Access-Control-Allow-Origin* and *Access-Control-Allow-Credentials* headers, respectively indicating the origins that are allowed to perform requests and whether cookies, HTTP authentication entries and client TLS certificates should be sent in subsequent requests or not. The allowed origins may be a wildcard * if credentials are not used. The response to a preflight request can include headers indicating the allowed methods, allowed request headers, response headers exposed to the requesting script and a max age before a new preflight request must be performed (default 5 seconds). The allowed and exposed headers may be a wildcard * if credentials are not used. [31]

It should be noted that CORS only protects requests originating from browsers. No protection is provided for malicious requests using other HTTP clients.

The *Cross-Origin-Resource-Policy* header can be used to forbid non-CORS requests that are either cross-origin or cross-site. Like CORS, it only provides protection against browser based requests.[31]

The *X-Content-Type-Options* header is used to forbid browser MIME type sniffing, and instead always use the MIME type provided in the *Content-Type* header [31]. The purpose of this is to prevent non executable MIME types from being converted to executable ones. Such conversion could cause XSS vulnerabilities. The header is mostly relevant for Internet Explorer which has a very aggressive MIME sniffing algorithm.

The *Content-Disposition* header is used to indicate whether the content of a regular HTTP request should be displayed inline or saved as an attachment [32]. The header is important for any API providing file downloads, as omitting it could allow the downloaded file to run untrusted code. A similar header, *X-Download-Options*, provides the same functionality for Internet Explorer.

Another important security header is *Content-Security-Policy* (CSP). It allows a website to whitelist domains that are allowed as sources for different types of content, and set some other request related security settings[33]. However, setting *frame-ancestors* to *'none'* helps preventing drag-and-drop based clickjacking attacks. The *X-Frame-Options* header can also be set to *DENY* for the same functionality in older browsers [34].

CSP can also be used for defense in depth by setting *default-src* to *'none'* to prevent further resource requests if a response was erroneously being displayed inline. With a similar motivation, *Feature-Policy* (recently renamed to *Permissions-Policy*) can be set to *'none'* and *Referrer-Policy* to *no-referrer*.

The *Strict-Transport-Security* header requires future communication with the origin of the request to be preformed over HTTPS. The header is ignored if sent over unencrypted HTTP[35]. The header is useful to prevent MITM attacks on unintentional HTTP requests.

The fetch metadata request headers is a set of headers sent by browsers that provide four request attributes that the server can use to decide if a request is allowed. This allows an API to implement a resource isolation policy, preventing unintended usage of the API. The first header, *Sec-Fetch-Dest* provide information on the destination of a request (different types of HTML tags, and a special empty type for other destinations such as fetch). The second header, *Sec-Fetch-Mode* provides

information on the request mode (navigation, same-site, CORS, no-CORS and websocket). The third header, *Sec-Fetch-Site* provides information on the same-site status of the request. It can be either same-origin, same-site, cross-site or none. The final header, *Sec-Fetch-User* indicates whether or not a navigation request was triggered by user interaction.[36]

The fetch metadata headers are not considered CORS safe[31]. The headers must be included in the allowed request headers of the preflight response in order to be sent by the client in the subsequent request.

2.9 Authentication

Authentication is the process of verifying the identity of a client. There are multiple authentication methods for HTTP requests. Authentication is complicated by the largely stateless nature of the HTTP protocol. API authentication often uses the `Authorization`[1] header. The header supports a number of authentication methods, such as basic authentication with username and password, digest [37], and bearer token [38]. Credentials should never be sent in plain text, and thus secure HTTP authentication requires HTTPS.

Traditionally, session cookies have been a common authentication method, but *Json Web Token* (JWT) is a newer method of authentication. Unlike tradition session cookies JWTs contain information about the associated user. JWTs are a Base64url encoded strings that contain a header, a payload and a signature. The header identifies the type of token (which is JWT), and the algorithm used to generate the signature. The payload contains the actual identification data of the JWT, and the signature is either a HMAC or a digital signature. The rationale behind JWTs is that authentication can be provided by another trusted server by validating that the signature was generated using a known cryptographic key. JWTs cannot be invalidated once issued and therefore support setting a time of expiry. Explicitly invalidating a JWT requires keeping a list of invalidated and not yet expired tokens on the server. [39] OpenID Connect (OIDC)[40] is an authentication framework built on top of OAuth 2 [41], which uses JWTs for authentication.

2.10 Authorization

Authorization is the process of determining if an subject has permission to perform an action on an object. The subject may be an API user, the object a REST resource and the action a GET request. A central concept in authorization is the principle of least privilege. According to this principle, a user should never have access to a resource they do not need to have access to. Thus, they should only have access to the minimal set of resources that are necessary to carry out their tasks. [42]

Role-based access control (RBAC) is an access control scheme where users are granted access to resources based on their role in an organization. [43] For example, a machine operator does not require access to order planning functionality. RBAC is an access control scheme in widespread use, but is inadequate to solve all authorization situations. [44]

Relationship-based access control (ReBAC) is another access control scheme. ReBAC grants users access to a resource based on their relationship with it. ReBAC originated in social network, where user data may be available based on friendship status [45]. A resource may for example only be editable by the employee who created it and the employee's manager.

Attribute-based access control (ABAC) is an access control scheme where users are granted access based on attributes of the subject, object and environment[46]. Management of ABAC permission policies can be significantly more complicated than other access control models. [47] Access may for example be restricted by the users department, and the time and location of the action. Roles and relationships may also be included as attributes in ABAC, effectively making it a superset of RBAC and ReBAC [47][48].

2.11 Model Driven Development

"Model-driven development (MDD) is a software-engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle." [49] MDD is a sub-field of Model Driven Engineering focusing model driven approaches to software development. In addition to the model of the application, a meta-model may be used to describe the modeling facilities available to the model (types of objects, relationships etc.) A meta-model specific code generator is used to translate the model into executable code.

Previous research has pointed out that MDD is most effective when used to model only parts of an application. It is generally more practical to implement certain functionality directly. MDD has also been most successful when using a domain-specific meta-model that capture the particular complexities of that domain.[50]

One aspect of MDD that is attractive from a security standpoint is the reduced risk of programming errors with a higher level of abstraction. In [51], it is found that practitioners of model driven development find quality and safety to be among the benefits of the approach.

2.12 Model Driven Security

A model driven approach can also be taken to security aspects. Model Driven Security (MDS) is a specialized MDE approach for developing secure systems. The majority of MDS research has been around authorization and confidentiality[52]. In particular, an MDS approach to authorization is discussed in [53]. In the paper, a Role Based Access Control (RBAC) scheme is developed based on an extension to UML named SecureUML. In addition to RBAC they implement authorization constraints that depend on application state, implementing simple ReBAC functionality.

2.13 Using MDD to build a REST API

Multiple approaches to generating REST API backends from models have been presented. One approach [54], models the backend based on REST concepts: resources and properties. The meta-model maps HTTP verbs to CRUD activities, and also includes functionality for database searching and wrapping of third-party functionality. The meta-model also includes support for hyperlinks between related resources. The meta-model includes definitions for HTTP basic authentication, but no other authentication options and no authorization functionality. The authors argue that the approach greatly facilitates REST development.

Another meta-model is presented in [55]. This meta-model splits resources into different types, among them primary resources, sub-resources, lists and filters. The proposed meta-model si split into a structural and a behavioral model, where the structural model again contains concepts such as resources and properties, while the behavioral model is centered around state transitions.

2.14 OpenAPI as the base for a meta-model

OpenAPI provides sufficient information to generate client libraries for interfacing with an API. Similarly, there exists tools for generating boilerplate server code without providing an actual implementation. One such tool with support for multiple backend technologies is *Swagger Codegen*. There also exists research on generating website clients from OpenAPI definitions using model driven development[56]. OpenAPI has also been integrated with UML modeling tools [57]

As OpenAPI can be extended using properties beginning with `x-`, it can not only be used to define a REST interface, it can also be used as a basis for a meta-model for an API backend by adding properties describing implementation details. In [58], the OpenAPI specification is extended using `x-swsg-ac`, `x-swsg-cc` and `x-swsg-name` properties. The proposed meta-model allows functionality to be defined using atomic components (`x-swsg-ac`), which can be written in any language of choice, as well as composite components (`x-swsg-cc`) that constitute a set of other components executed in sequential order. The `x-swsg-name` property is used to link an OpenAPI method definition to a component. The article also discusses how to validate consistency between the inputs and outputs of the components and the method definition.

2.15 Oracle Database

Oracle database is an SQL database that ABB MES relies heavily on. A short review of the functionality of the database is presented in this subsection, as it is heavily referenced when discussing the implementation of the REST API backend.

Oracle DB, as any other relational database is built around the concept of tables. Tables have a fixed set of columns, and each column contains a particular property of the object the table represents, and has a fixed type. A table can contain multiple data instances known as rows. A row has a value for every column in the table. [59]

Oracle DB supports a number of data types. The main string type is `VARCHAR2`, which supports strings of variable length up to a predefined maximum length. Other string types include the fixed length (in the sense that it always uses the same amount of storage space) `CHAR` and the arbitrary length `CLOB`. There are also variants for national character sets, `NVARCHAR2`, `NCHAR` and `NCLOB`. Additionally, there is a legacy `LONG` data type for strings of arbitrary length. [59]

Numbers can be represented using the `NUMBER` data type. This number type is for fixed point numbers, and each column has a precision (number of decimal digits) and a scale determining the position of the decimal point. Numbers with a scale of 0 (or less) implies that the number is an integer. The type `FLOAT` is a `NUMBER` with scale determined by the contained data. Finally `BINARY_FLOAT` and `BINARY_DOUBLE` are 32-bit and 64-bit floating point numbers respectively. There exists no type for booleans, and they are thus defined as numbers with precision 1. [59]

Dates with second precision without timezone are represented by `DATE`. There are three more precise date formats that include fractional seconds, `TIMEZONE`, `TIMESTAMP WITH TIMEZONE` and `TIMESTAMP WITH LOCAL TIMEZONE`, which vary in their treatment of timezones. [59]

Finally there exists a number of types for binary data. `BLOB` stores binary data in the database and `BFILE` provides read access to external files. Additionally there are two legacy types for binary data: `RAW` and `LONG RAW`. [59]

Oracle DB can execute code written in the procedural PL/SQL language to perform more advanced operations than possible in SQL. PL/SQL can be used both directly in queries, and to define stored procedures and functions. The difference between procedures and functions is that functions have a return type. The parameters of both procedures and functions can however have both in and out direction (as well as both), allowing procedures to return data. Functions and procedures can be grouped into packages. [59]

The connection between Oracle DB and clients is always encrypted and clients by default authenticate using a username and password. Oracle DB also supports certificates for authenticating the database and mTLS to provide both client and server authentication. [59]

3 Implementation

This section describes the implementation of the REST API backend for ABB MES. In particular, it focuses on the implementation of security related aspects, and how security was taken into account while developing the functional aspects of the backend.

The first subsection contains an overview of the legacy backend and frontend, including relevant implementation details and security model. The second subsection discusses the requirements for the new backend, in particular its security requirements. In the third subsection, the choices of technologies for implementing the backend are presented and in the fourth subsection motivations are provided for using MDD. The fifth subsection presents the basis for the used meta model. Subsections six through nine go through the parts of the model, specifying functional extensions, validation,authorization and mappings for data types. The tenth subsection focuses on the process of deriving the model for the endpoints. It includes descriptions of both automated and manual steps in the process. In the eight subsection the code generation process is described, as well as the security aspects of the generated code. This is followed by the parts of the application not using MDD, namely HTTPS, authentication and logging.

3.1 Legacy ABB MES backend and frontend

ABB MES is a Manufacturing Execution System for the pulp and paper industry. It implements functionality for production planning, production management, quality management, decision support, energy management and order management. This functionality is logically divided into multiple applications. ABB MES is built using a client-server architecture where Oracle DB is used as database. The client-server communication uses a proprietary protocol. There exists both desktop and browser-based clients for ABB MES. The client UI is organized into separate programs, that correspond to UI views implementing a particular form of functionality.

Most of the applications are implemented by means of a fat server-thin client model, where business logic runs in the server and the client is only responsible for displaying the UI. A few applications however are implemented as fat clients with business logic mainly running client-side.

ABB MES heavily utilises PL/SQL code for business logic. Some functionality is implemented using stored procedures, functions and database triggers. It is noteworthy that multiple tables have custom ID columns that must be generated using procedures or functions. The backend utilizes a custom form of code referred to as transactions for communicating with the database and clients. Transactions are made up of steps, and each step either executes PL/SQL code or sends UI instructions to the client. Note that this PL/SQL code is separate from that stored in the database.

Authentication in ABB MES is implemented either using local users or Windows Active Directory (AD) users. Both authentication models authenticate the user based on username and password. Both types of authentication are accessible through the

database using procedures. Sessions last for the duration of a connection, and each session has a separate database connection. There is currently an undergoing effort to transition to using an OpenID Connect based authentication solution.

ABB MES uses a RBAC permissions model. Permissions are assigned to groups that users can be members of. Thus groups map to roles and users to subjects. Permissions are assigned with program granularity, and each program has separate permissions for create, read, update and delete actions.

User input is validated according to the UI elements data is inserted into. Data is not validated at the database end based on check or foreign key constraints.

3.2 Requirements for the new backend

There are a number of requirements, both functional and security related that the new backend must fulfill. There are concerns relating to API design, interoperability with the legacy system, authentication, authorization and request data validation.

The main driver for implementing a REST API for ABB MES was the customers interest of integrating with the system and extracting data from it. Thus, the main interest was in the data available in the database, not access to the business logic. For this reason, the new backend should interact directly with the database, and not go through the existing UI-oriented backend.

A REST API should be built around the concept of resources identified by paths, and the manipulation of said resources. The simplest approach to exposing SQL data through a REST API is to view tables as resource types and individual rows as resources. This allows us to expose row-oriented CRUD endpoints that map to the HTTP methods GET, POST, PUT, PATCH and DELETE. In fact, tables can also be considered resources that gather the individual rows into an array and allows creation of new row resources. Thus, table-oriented resources should provide GET endpoints for enumeration of the set of all rows and POST endpoints for the creation of new rows. Meanwhile, the row-oriented resources should provide GET endpoints to fetch an individual row, PUT and PATCH endpoints to modify an existing row and DELETE endpoints to delete an existing row. Both types of GET requests should always be idempotent, as intended by the HTTP specification. Listing 1 shows these endpoints for an example table.

```
GET    /api/v1/diary/items
POST   /api/v1/diary/items
GET    /api/v1/diary/items/000000001
PUT    /api/v1/diary/items/000000001
PATCH  /api/v1/diary/items/000000001
DELETE /api/v1/diary/items/000000001
```

Listing 1: Example of endpoints associated with a table

The GET endpoint of a table-oriented resource cannot necessarily return all rows in a table in the case of large tables. For this reason the returned data must be paginated,

with a limited page size to mitigate denial of service attacks. Setting a smaller page size should be possible, for cases such as visualizing the data in a table or chart. It should also be possible to filter the rows based on the columns of the table. This should be possible by means of query parameters. Filtering should not only be possible based on equality, it should also be possible to filter e.g. based on date ranges. An example of a filtered query could be `/api/v1/diary/items?day=2022-12-17&shift=1`.

Additionally, the rows in some tables can be considered to be owned by certain rows in another table. In these cases the table-oriented resources of the owned table should be considered to be sub-resources of the row-oriented resource of the owner. This means that the owned table-oriented resource should be filtered for each row of the owning resource, the table-oriented resource would then have a path such as `/api/v1/diary/items/000000001/comments`. In some cases a table may be viewed as being owned by two or more separate tables. When this is then case, a resource may be exposed through multiple paths. If two tables are related, but the relationship cannot be considered an ownership relation, neither resource can be considered a sub-resource of the other. In this case, no path based filtering should be provided. Instead, filtering may be performed based on query parameters.

POST requests should contain all data necessary to insert a new row. The endpoint must be capable of generating a new id for every inserted row, if required. It should also return a copy of the row in the response on successful row creation. We recognize that all columns should not be directly settable. Some of these columns are set using database triggers, but the endpoint must be capable of computing the correct values for any columns not set by triggers based on the request data.

PUT requests should always include all directly settable columns. They must not include any additional data. PATCH requests should include at least one settable column, and are allowed to omit any settable column. Columns whose values are computed by the backend must be recomputed if any data they depend on is included in the request. Both POST and PATCH endpoints should return the entire modified row.

DELETE requests should not contain any data beyond the id of the resource to be deleted. The deletion of one resource with multiple sub-resources should generally lead to the cascading deletion of all sub-resources. For simplicity we require that all sub-resources be deleted before the deletion of such a resource. This relationship is based on the paths, any resources below a particular resource in the path is considered a sub-resource. More advanced deletion behavior is not supported by this simple model. DELETE endpoints should always return an empty response on success.

There exists functionality in ABB MES that does not map well to the API design described above. Modification of one table may also require modifications to another table. For example, if a paper roll is moved from one machine to another, this should be reflected in its history, stored in another table. On the other hand, the rolls history should not be directly modifiable. The correct behavior in these cases already exists in the existing code base, either in the form of a stored procedure or a transaction. It should thus be possible to perform modification of tables through these existing procedures and transactions, but they should still return the entire modified row as usual. In the case that these do not map exactly to CRUD operations, additional

endpoints should be exposed on sub-paths of the concerned resource. One example of such a path could be `/api/v1/core/units/00000001/move` These paths should only provide the POST method, as it is the method that best corresponds to an arbitrary action that modifies the state of the resource. This behavior is not in line with REST, as operations on a single resource are spread out over multiple paths, but is included as a pragmatic approach to increase compatibility with the existing system.

In addition to tables, ABB MES also includes table functions that derive their returned data from tables. Table functions should also be exposed as resources. These resources should not allow modification of data, as the returned data already belongs to some table. The parameters of a table function can be supplied either as path or query parameters, depending on if their semantics identify the resource or provide additional parameters. Normal functions can also be used to fetch data. In these cases the semantics of the corresponding endpoint aligns with that of a GET of a row-oriented resource. Care must be taken so that table functions and normal functions do not break the idempotency requirement for GET methods.

All request data must be properly validated. Validation logic from the legacy backend cannot be used as it is centered around the UI of the client. There is also no database validation to rely on. If invalid data is stored in the database, it may affect the functionality of the legacy system. Special care will have to be taken, as the constraints for the data may not be immediately obvious. It must be possible to validate both static constraints, like a numeric value that must be positive, and dynamic constraints such as foreign keys and value ranges defined in the database.

A concern related to validation is sanitation of request data. Inclusion of the common web languages HTML or Javascript should not be allowed in string columns. Additionally, care should be taken when constructing SQL and PL/SQL statements to not include user supplied data, as this could open up for SQL injection attacks.

No actions should be allowed to be performed by unauthenticated users. Authentication should use the same facilities as the legacy client, and only AD based authentication should be possible. The new backend must be responsible for maintaining sessions after login, and provide logout functionality. The sessions should allow only a limited time of inactivity, and it should be possible to explicitly log out from a session. The new authentication system should be flexible and allow for Open ID connect authentication in the future.

The backend should check that a user is authorized to access a resource before granting access. The program centric permission model of the legacy frontend does not map well to a REST API with resource centric endpoints. However, the users and groups of the legacy system should be reused as these form a good foundation for RBAC. There should be a separate permission for each endpoint to allow for flexibility when assigning permissions. The permissions should be hierarchical and support wildcards, allowing for example all methods of one resource to be accessed with a single permission definition. In addition, users should only have access to limited set of rows in certain tables based on their relationship to the data. This includes tables where the user should only be able to access rows created by themselves. Endpoints may in this case provide an additional high privilege permission that can access all

data. This provides simple ReBAC functionality. No ABAC functionality should be provided at this point, as it would require significant alterations to the existing authorization model.

The REST API should require the use of HTTPS for all communication. Encrypted communication ensures that untrusted parties cannot eavesdrop on the communication or alter it. The use of certificates also mitigates MITM and impersonation attacks. Only HTTPS with TLS 1.3 should be supported.

The REST API backend must be able to fetch data using two methods: directly from the database and through database procedures. Likewise, it must support updating data both directly to the database and through procedures. Additionally, it must be able to use procedures to calculate the values of some columns based on the data received from the user.

A related concern is that of foreign keys pointing to data in different tables. ABB MES does not make use of OracleDB's built in support for handling foreign keys, and thus the backend must take care of this matter. Before any column that is a foreign key is inserted or updated, the existence of the new value in the target table must be verified.

The API should use JSON as the format for both request and response data. Supporting a single type of data and not e.g. XML simplifies the implementation of the API. Requests containing any other form of data should be rejected.

3.3 Technology choices

The REST API backend was implemented with the architecture shown in figure 1. It includes the Oracle DB of the legacy application, a reverse proxy, a Redis cache and a number of instances of the actual API backend. This thesis mainly focuses on the backend component.

The main component of the backend was implemented in Node.js using the *express* web server. There were no security considerations behind these decisions, but rather it was based on the developers familiarity with the technology and the availability of third party components interfacing with these technologies. Authentication was implemented based on the *passport* library, which offloaded some tasks such as the verification of JWTs. It should be noted that passport by itself does not restrict access for users that are not logged in.

The library *express-openapi* was a central component used in the development of the application. It provides support for building express apps based on OpenAPI definitions. It takes care of concerns such as routing and input validation based on the OpenAPI specification. It also provides support for tying into the authentication methods defined in the OpenAPI definition. In addition, it validates the OpenAPI definition used, and allows for specification of validation for extensions to OpenAPI. It does not validate the presence of any additional parameters beyond those defined in the OpenAPI definition.

It was decided not to use an ORM, as we needed to be able to use PL/SQL combined with database queries, for example to compute ids for new rows. Instead, we opted to use *node-oracledb*, which provides the official bindings of OracleDB for

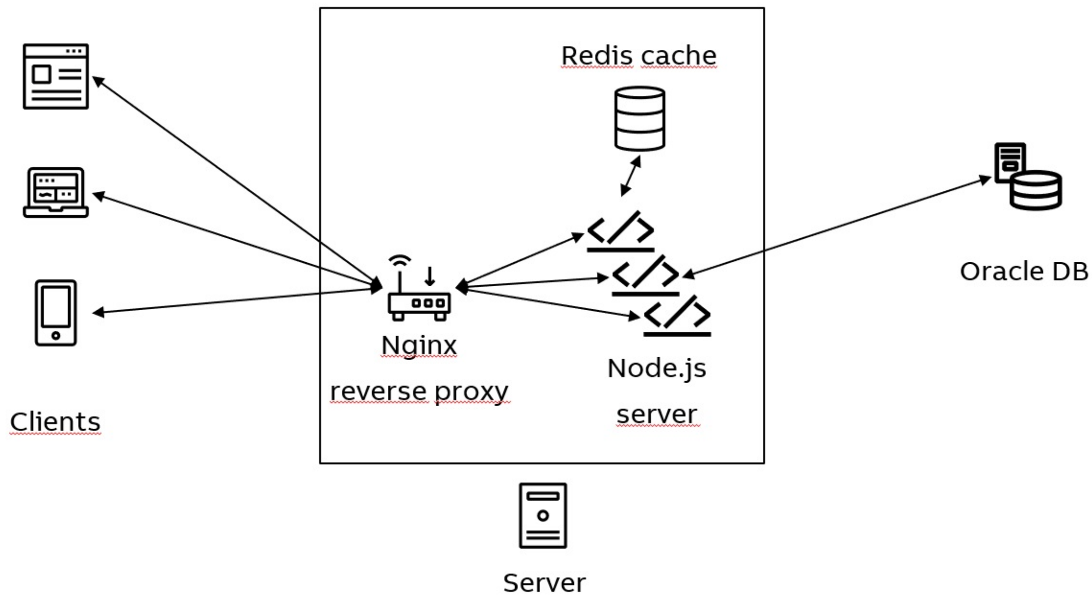

Figure 1: Architecture of REST API backend.

Node.js. This requires us to write SQL statements ourselves, and potentially opens the backend up for SQL injection. To prevent SQL injection, we used prepared statements and bind variables to insert data into the query. The data flow diagram in figure 2 gives an overview of the request handling process.

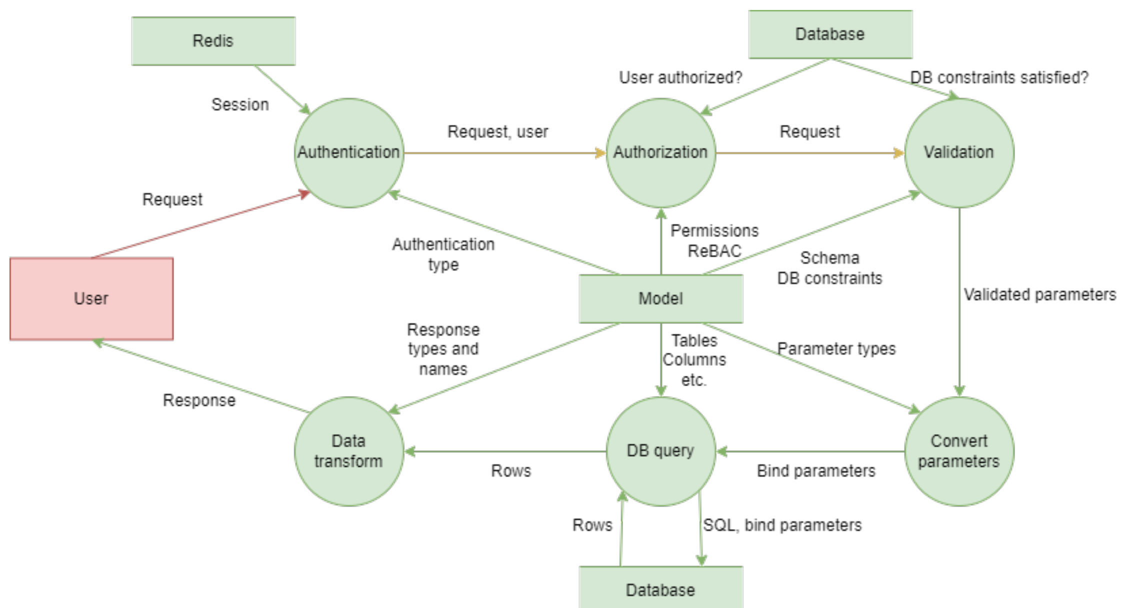

Figure 2: Data flow diagram of the REST API backend.

For storage of active sessions the *Redis* cache was used. It allows sessions to be

stored for a limited amount of time, and this time can easily be extended every time a new request is received, allowing for simple implementation of session timeout.

3.4 Motivation for MDD

The legacy frontend accesses the database using PL/SQL transactions. These are generally developed for the needs of a particular view (they may not e.g. fetch all data in a table). Thus, they are not well suited for the implementation of a generic API. It was decided against using transactions to implement the backend, due to the above mentioned fitness concerns, and the fact that it would have required a custom mechanism for converting transactions into usable PL/SQL. There are then two remaining methods of accessing the data in the database: direct SQL queries and PL/SQL procedure calls. This limited set of actions is well suited for MDD, and allows us to use a narrow meta-model to describe the functionality of the backend. Note that we will be modeling the relationship between REST endpoints and database constructs, not the business processes of a paper mill, as this is already accomplished by the existing system.

The raised level of abstraction of MDD reduces the risk of coding errors in the implementations of the endpoints. In our case it is particularly useful as we have a potentially large number of endpoints. It is also possible to generate a large portion of our model based on the existing definitions in the database, further reducing the risk of error. There are however a number of concerns remaining. The database does not contain any constraints for inserted data, not even for foreign keys, so these constraints have to be included in the model manually. We also note that MDD is not fit to implement every aspect of a REST API backend.

3.5 Domain specific meta-model based on OpenAPI

We decided to use OpenAPI as a basis for our meta-model. OpenAPI provides a framework for defining REST endpoints, including static constraints for parameters. There was also a requirement for documentation in the form of an OpenAPI definition, which would simultaneously be fulfilled. There is also a large number of existing frontend software components interacting with OpenAPI definitions.

The meta-model presented in [58] is very generic and allows functionality to be defined as individual atomic functions that are built together to create composite functions which can handle requests to an endpoint. This approach was not chosen, as we are interested in a very specific domain: defining a REST API centered around tables and rows in a relational database. Thus, it was decided to instead develop a a meta-model centered around these concepts.

OpenAPI however does not provide all the functionality we require. It has no way of encoding relationships between REST resources and the underlying database. It also lacks the ability to encode dynamic constraints that depend on the contents of the database (including foreign key constraints), as well as authorization parameters.

OpenAPI allows us to define arbitrary JSON structures for both request and response data. This is too generic for an API that is centered around tables and

rows in a relational database. It was decided to limit all HTTP body data to flat JSON objects, or arrays of flat JSON objects. This allows us to have a tight coupling between objects and rows. Arrays of objects are used only in the responses to GET requests to table-oriented resources, as these return multiple rows. All other methods return objects and request body data is always an object. The rest of this thesis will use the terms *request body property* and *response body parameters* to refer to the individual properties of the JSON objects in request and response data respectively.

Another restriction that was put on the OAS was that all parameter names must be unique. A single method must not take path parameters, query parameters, or request body properties with the same name, even though OAS allows this. This is to ensure simplicity when defining bind parameters for database communication, as well as validation and computation logic.

3.6 Functional extensions

OAS required extensions to interact with a relational database. In particular, mappings were required for tables and columns. We decided to map a pair of OpenAPI paths to every table: one for queries to the entire table and row insertion, and one for modification of a particular row. These types of paths had to be discernible from each other in the meta-model. For this purpose, a `x-resource` property was added to each path, with a value of either `row` or `table`. The table that the endpoint maps to was identified by a `x-table` property of the path.

Paths with a `x-resource` value of `table` were only allowed to have GET and POST methods defined, while paths with a value of `row` were allowed to have GET, PUT, PATCH and DELETE methods. This restriction was imposed to adhere to the requirements for table-oriented and row-oriented resources specified in 3.2. Additionally, the GET method of `table` resources must have a response with status code 200 OK and a JSON array of flat objects as its body, while the POST method must have a request body with a flat JSON object and a response with status code 201 Created and a flat JSON object as its body. The GET, PUT and PATCH methods of `row` paths must have a response with status code 200 OK and a body with a flat JSON object. The PUT and PATCH methods must have a request body with a flat JSON object, while the GET method must have no request body. The DELETE method must have no request body, and a response with status code 204 No Content and no body. An example of a `row` path can be seen in listing 2. All response body properties must be required, except for the PATCH method, where no property can be required.

The request and response body properties must have a mapping to the corresponding column in the table associated with the path. This relationship was encoded using the `x-column` property of the schema of the property. Any columns that should be set programmatically, and not directly by the user must be set to read only, or be omitted from the request body schema. An example of a property definition is provided in listing 3.

Only GET methods of the `table` paths were permitted to have query parameters. These query parameters were used for filtering the rows returned by the method. If

```yaml
/api/v1/diary/items/:itemId:
  x-resource: row
  x-table: ITEM
  parameters:
  - name: itemId
    in: path
    ...
  get:
    responses:
      '200':
        content:
          application/json:
            items:
              type: object
              ...
  put:
    requestBody:
      required: true
      content:
        application/json:
            schema:
              type: object
              ...
    responses:
      '200':
        content:
          application/json:
            schema:
              type: object
              ...
  delete:
    responses:
      '204':
```

Listing 2: Model for a `row` path with GET, PUT and DELETE methods.

multipe query parameters were specified, the results should be filtered according to all parameters (and logic). The `x-column` property of the schema of the parameter was used to map the parameter to a column. By default, query parameters check for equality with the specified column. To allow for other types of checks, the `x-operator` property was introduced to the parameter definitions. Its possible values were '<', '<=', '>', '>=' and '='. This allowed for the creation of multiple parameters mapping to the same column with different comparison semantics. Path parameters were defined similarly, but always used equality comparison. An example of a query parameter definition is provided in listing 4

```yaml
itemId:
  type: string
  readOnly: true
  maxLength: 20
  x-column: ITEM_ID
```

Listing 3: Model for a property.

```yaml
name: dateGt
in: query
schema:
  type: string
  format: date-time
  readOnly: true
  x-column: ITEM_ID
```

Listing 4: Model for a query parameter.

Some columns should not be directly settable through the API, but rather calculated, possibly based on other values. A special category of these were id columns that identify an individual resource. These were calculated using PL/SQL procedures and functions, or set by database triggers upon row creation or modification. When procedures and functions were used for computing column values they were encoded using the `x-computations` property of the path. The `computations` property was an array of computations where each computation was an object with `name`, `parameters`, and for functions `return` properties. The `name` property specified the name of the procedure (possibly including package name), the `parameters` property was an array specifying the parameters of the function or procedure in the order of its definition, and the `return` property specified the return value of a function. The parameters were described as objects, with a `type` property. The possible values for `type` are `in` (a bind in parameter), `out` (a bind out parameter), `const` (a constant value independent of the request), `status` and `error` (special purpose parameters used in ABB MES procedures). The parameter types `in` and `out` additionally had a `name` property. For `in` parameters this was the name of a request body property, or path parameter to use as input for the computation, while for `out` parameters it was the name of a column to store the result in. The `const` parameter type has a `value` property for a constant value. The return property was a string with the name of a column to store the return value in. It is noteworthy that this model does not allow the results of one computation to be used as inputs for another. An example of the `x-computations` property is given in listing 5.

As all functionality could not be implemented with these two resource types, the meta-model also included the option of defining custom resource types. This was used to support procedure and function based resources. It was also used to implement the login and logout endpoints.

```yaml
x-computations:
- name: ID.DIARYITEM
  parameters:
  - type: in
    name: date
  - type: out
    name: ITEM_ID
  - type: status
  - type: error
```

Listing 5: Model for computations.

3.7 Request data validation

A requirement for the REST API backend was that both static and dynamic validation should be supported. OpenAPI already provides facilities for describing static constraints, where input is validated according to static values. It also support pattern based string validation using regular expressions. Extensions are still required for validation of dynamic constraints that depend on either database data or encode a relationship between two values in the request. A special case of database constraint is that of a foreign key.

All dynamic validation constraints were included in the `x-validation` property of the paths. This property was specified as an array of individual constraints, where each individual constraint was an object. Each constraint object had a `type`, which could be `property`, `foreign-key` or `database`.

Constraints of the `property` type were used to encode constraints between pairs of columns in the same row. These were specified using the names of the corresponding request body parameters and an operator. The property names were included in the `parameter1` and `parameter2` properties of the constraint. The operator was specified in the `operator` property, with the same semantics as `x-operator` in the previous subsection. The constraint should be understood as *from type to*, for example $start \leq end$.

Foreign key constraints were encoded using the `foreign-key` constraint type. This type of constraint included a `property` property, with the name of the property that should be validated. Additionally, foreign key constraints must include `table` and `column` constraints identifying the table and column to which the foreign key points. It is possible that a foreign key constraints points to the same table as the one that is modified, this is the case when the table encodes a hierarchical relationship. In this case, self-references may not be desirable. Therefore, foreign key constraints can include the `notSelf` boolean property to forbid self-references.

Generic database constraints can be encoded using the `database` constraint type. This constraint is useful in cases where configuration tables contain a single row, or set of rows, against which the request body property must be validated. It must have a `property` property. Again, this property specifies which request body property the constraint should validate. Additionally, the constraint must have a `table` and

`column`, respectively identifying the column and table against which the constraint should be checked, as well as an `operator` property specifying the comparison type. There was also a `where` property containing an SQL where clause allowing the results to be limited.

These three constraint types do not allow all possible types of database constraints to be validated. If additional forms of validation is required custom validation types can be specified. Examples of the standard dynamic validation types are provided in listing 6.

```
x-validation:
- type: property
  property1: startDate
  property2: endDate
  operator <=
- type: foreign-key
  property: itemId
  table: ITEM
  column: ITEM_ID
- type: database
  property: itemType
  operator: =
  table: CONFIG
  column: VALUE
  where: CONFIG_NAME = 'DiaryItemType'
```

Listing 6: Model for dynamic validation.

3.8 Authorization

The meta-model represents authorization data using the `x-authorization` property of path or method definitions. Authorization definitions on methods always override those on paths. The main purpose of providing separate permissions for methods is to use the `user` property described below. For example, editing of a resource could be restricted to a specific user while still allowing all users to fetch it. The `x-authorization` property is an array of authorization objects. Each authorization object must have a `permission` property containing a permission string. Table-level (function-level) authorization is only provided through this RBAC mechanism. An authorization object may also have a `user` property, specifying a column in the target table of the path that must contain the username of the current user. This property allows the implementation of rudimentary ReBAC, and is the only form of object-level authorization provided. This access control model does not make it possible to define access to a sub-resource based on a parent resource.

The methods of a path were mapped to the separate CRUD permissions provided in the authorization model of the legacy system. PUT and PATCH both mapped to the update permission, as they provide two variants of the same functionality.

There is no functionality included in the meta-model to implement column-level access control. If such restrictions are required, it is possible to include separate paths in the model for the same table, exposing different columns in each path. Implementing column-level access control for the same path would have required changes to the restrictions on response definitions, and added additional complexity to the implementation of the request handlers. An example of an authorization definition is provided in listing 7.

```yaml
x-authorization:
  # Standard permission, access to rows where the user is author
- permission: api.diary.item
  user: AUTHOR
  # Administrative permission, access to all rows
- permission: api.admin.diary.item
```

Listing 7: Model for authorization.

3.9 Mapping between OracleDB and OpenAPI data types

The main data types used in the database are `NUMBER`, `VARCHAR2` and `DATE`. These do not have a one to one mapping to the corresponding OpenAPI types `integer`, `number` (floating point number), `boolean` and `string`. Additional properties must be considered to construct a bidirectional one-to-one mapping. The defined mappings are presented in table 1.

Database type	Precision	Scale	OpenAPI type	OpenAPI format
NUMBER	any	> 0	number	-
NUMBER	> 1	≤ 0	integer	-
NUMBER	$= 1$	$= 0$	integer OR boolean	-
FLOAT	any	-	number	-
VARCHAR2	-	-	string	-
DATE	-	-	string	date OR date-time

Table 1: The mappings between Oracle DB and OpenAPI types.

`NUMBER` maps either to `integer`, `number` or `boolean` in OpenAPI. Which type to use can be determined based on the scale (number of digits after the decimal point) and the precision of the database type. If the scale is non-zero the correct type is always `number`. Otherwise, if the precision is greater than 1 `integer` is always the correct type. If the precision is 0, the correct type may be either `integer` or `boolean`, and manual inspection is required.

`VARCHAR2` always maps to `string`.

JSON does not have a special type for dates. Thus, dates have to be sent as strings in RFC3339 format. The `DATE` type may correspond to the OpenAPI `string`

type with either `date` or `date-time` format. Which format should be used cannot be determined based on database definitions.

There are a number of other data types used in a limited number of tables in the database. These can be categorized as alternate string, date, and number types, and types for arbitrary data. The string types are `CHAR`, `NVARCHAR2`, `CLOB` and `LONG`. The date types are `TIMESTAMP WITH LOCAL TIME ZONE` and `TIMESTAMP`. The only number type is `FLOAT`. The types for arbitrary data are `LONG RAW`, `BLOB` and `ANYDATA`. Of these types, only `FLOAT` is supported as it is an alias for `NUMBER`. All other types require special handling if exposed.

3.10 Deriving the OpenAPI model from existing database structures

OracleDB contains specialized tables storing information on tables, indexes, columns and procedures. This data was exploited to provide a foundation for the OpenAPI model and reduce the need for developer provided input and its associated risk of mistakes. A tool was written for querying the database structure of ABB MES, and produce an initial OpenAPI definition.

As input, this tool took a list of tables to expose, and which CRUD operations to generate for each table. This was to prevent access to sensitive data and modification of certain tables.

Using the provided list of tables the tool mapped each table to two paths in the OpenAPI definition: one of type `table` and one of type `row`. Based on the CRUD operations to expose, GET and POST operations were added to the `table` paths, as well as GET, PUT, PATCH and DELETE operations to the `row` paths. The OpenAPI definitions were split into multiple files, with one file for each pair of paths. All parts of the OpenAPI definition that are described in the following paragraphs were located in the same file as the paths of the associated table. A master OpenAPI file was generated with references to each path.

The tool generated schemas for each column. The data types of the schemas were determined by the mapping described in 3.9. In the cases where the data type could not be exactly determined, a placeholder was inserted to indicate the need for developer input. For number columns, coarse maximum and minimum values could be determined based on the precision and scale of the database types. For string values a maximum length could be determined. Nullability of columns could also be detected using this method.

Each table in the database contained special columns for the time, program and user that created the row and last updated it. As these columns were set by a database trigger, they were always marked as read only. For other columns the read-only status could not be determined, except for id columns which were always read only.

Schemas were also generated for the request and response data objects. The same schema with all individual properties set as required could be used for both requests and responses, except for the PATCH request, which required all individual properties of the object to be optional. If the request body properties for creation

differed from those for update, this would require manual editing of the definition.

The path parameters were determined based on unique indexes, as no primary keys were defined in the database. For tables where there were multiple or no such indexes, a placeholder was inserted to indicate the need for developer input. All identified path parameters were set as read-only to prevent user modification.

The query parameters for the GET request of table-oriented resources were generated by adding parameters for each combination of type relevant comparison operators and columns. This added at most five query parameters for each column. The desired query parameters could be manually filtered after this stage.

Authorization definitions could partially be defined based on the database. Each table could be assigned a unique permission, but beyond this no permission hierarchy could be determined. It was also not possible to determine `user` properties for the authorization.

The values of a number of model parameters could not be determined from the database, and for some parameters the correct value could only be determined in some cases. Thus, a large portion of the model still had to be verified by developers responsible for developing these systems.

Validation constraints could only be defined coarsely based on data available in the database. These coarse definitions required tightening by analysing the semantics of their intended use case. For example, shift numbers could be limited to the integer values 1, 2 and 3. This also concerned only static validation. Dynamic validation constraints could not at all be determined based on the database. All dynamic validation thus had to be inserted manually into the model.

Another issue was that of which columns should be manually editable and which should be computed. The computation method could not be derived from the database and always had to be specified manually. Conceivably, the parameter and return types of procedures and functions could have been automatically extracted from the database, but this would have been of limited utility as the mapping to request parameters and intended use case still would have required manual specification.

It was not possible to determine any relationships between tables. The path templates thus required manual editing to organize them into a logical hierarchy. It was also not possible to determine which columns referred to parent resources, and thus not possible to include these in the path parameters of both row-oriented and table-oriented resources.

3.11 Code generation and implementation

The REST API backend did not include traditional code generation that produces source code files for inclusion in the project. Instead, it relied on Javascript closures to pass data included in the model to the operation specific handlers. Generic handlers were written for each type of supported endpoint (GET and POST for table-oriented resources and Get, PUT, PATCH and DELETE for row-oriented resources), as well as enclosing closures that derived the parameters required for the request as exactly as possible during application initialization.

For most request handler types, the PL/SQL required could be completely derived

from the model, and thus did not depend on user input. This was possible as they always take exactly the same parameters and return exactly the same type of data. The only exception to this was the GET method of table-oriented resources and PATCH method of row-oriented resources. Although they return the same type of data for all requests, both take a variable number of parameters. The GET method uses query parameters to allow filtering of the returned rows, and thus requires a dynamic where clause. The PATCH method, on the other hand, allows updates to an arbitrary subset of columns, requiring a dynamically defined list of columns to update. The PATCH method also requires database based validation and column computation to be dynamic, and both of these require PL/SQL to implement.

Certain parameter types required special handling. Booleans were stored in the database as numbers, and thus required conversion both from parameters to bind variables and from returned database rows to JSON for the response. Dates similarly required special handling as they had to be serialized to strings to be sent over HTTP and node-oracledb required them to be Javascript dates.

All string input was sanitized for HTML content, to prevent XXS attacks against clients. This was accomplished using a third party library. Which properties were strings requiring sanitation was determined based on the schemas in the OpenAPI definition.

3.12 HTTPS

HTTPS was implemented through the reverse proxy. It was configured to only allow TLS 1.3 connections, to protect against existing vulnerabilities of older versions of the protocol. All other settings relating to HTTPS were set to their defaults. Some HTTPS services provide a HTTP endpoint to redirect accidental unencrypted requests to HTTPS. We opted not to include such functionality, as the HTTP connection is susceptible to MITM and impersonation attacks.

3.13 Authentication

As we needed to integrate the authentication process with the existing authentication method for the legacy client, the authentication procedure was implemented within the new backend instead of relying on a separate authentication solution such as OIDC. The authentication system uses password based login and relays the credentials to the database which verifies them. This system will be replaced by OIDC in the future.

For session authentication token based authentication was chosen. In particular, JWTs were used as the token. The tokens were sent in the `Authorization` header as this was deemed more secure than cookies or query parameters. The tokens were not sent in the body as this would have violated the REST constraint of uniform interface, including authentication information in what should be a representation of a resource.

Explicit logout support was included by storing every issued JWT in a Redis cache. The JWTs stored in the cache were given a limited lifetime and were refreshed

every time the authenticated user made a request, leading to an implementation of sessions permitting a limited amount of inactivity.

Accesss to the login endpoint was rate limited, both per user and per IP, to prevent both credential stuffing and brute force attacks.

The JWTs were verified using a HMAC, as the backend currently is responsible for both issuing and verifying tokens, leaving no need for public key cryptography. The JWTs were kept small to limit the amount of data included in the request. The information included in the JWTs was issuer, audience, subject and issuance time. No expiration time was included, as the backend regardless is required to maintain sessions to support explicit logout.

3.14 Logging

Request logging was implemented using the third-party *morgan* library. All requests were logged, including their responses and the issuing user, if authentication information was present. Thus both regular access, authentication and authorization information was logged, by parsing the HTTP status code. No body data was logged. In addition to requests, server errors and authentication related events such as lockout due to rate limiting were also logged.

3.15 Security headers

CORS support was added to the new backend by means of the *cors* package. By default, the package allows requests from any domain and does not impose any restrictions on the headers sent from the client. We configured the package to only allow requests from trusted domains and to not allow any additional headers, as these are not used by the backend. Additionally, no response headers were exposed to the client code, as the backend does not use headers that the code requires access to. No max age was set, leaving the default 5 second max age in force.

We note that all browser initiated requests made to the backend require preflight. All normal endpoints require the inclusion of credentials in the request in the form of a session cookie. The only exception is the login endpoint, which requires preflight only due to its use of JSON as the content type of the body. A layer of protection against CSRF attacks is provided by only allowing requests from whitelisted domains. As an additional protection CORB is set to same-origin to forbid any no-CORS requests from browsers.

Support for the fetch metadata headers was included to prevent potentially malicious requests. The only allowed destinations was 'empty', the only allowed mode 'cors' and the only allowed sites same-site and cross-site. This both adds an additional layer of protection against CSRF attacks on top of CORS, and prevents all same-origin access to the API. Such access should not even be possible as the backend never renders HTML responses, but the headers are used as a defense in depth measure. A 400 Bad Request response is returned if these checks fail.

MIME type sniffing could lead to XSSI attacks if the browser incorrectly determines the response data to have an executable MIME type.

One requirement for the backend was to only function over the encrypted HTTPS protocol. To prevent any downgrade to HTTP the HSTS header was set on every request. Additionally the XCTO header was set to `nosniff`, to prevent MIME type sniffing.

The database includes creation and last update times for every row. We exploited the last update time to implement the `If-Unmodified-Since` HTTP header. The header was required whenever updating or removing any row. This improved the integrity of the database by preventing updates based on stale data.

4 Results

The implementation of the REST API backend described in the previous section takes into account many security considerations of developing a REST API backend for a legacy system. Still, there are numerous issues that could be improved. The main reason such improvements were not made were time constraints and compatibility with the legacy system.

The backend uses the HTTP server included in Node.js. This leaves the handling of the HTTP protocol as the responsibility of a third-party component. This is common practice when developing any form of web application, and it speaks to the complexity of properly handling HTTP at the protocol level. However, this still leaves the REST API backend open to any potential vulnerabilities in the underlying software.

This section goes through the results of the thesis. It starts with an evaluation of the use of MDD, which is followed by a security analysis using OWASP API Top Ten, HTTPS properties and common browser vulnerabilities. Finally, it goes over special considerations found for legacy systems and remaining areas of security improvement.

4.1 Evaluation of the use of MDD

The use of MDD reduces the risk of security related mistakes, however this is only the case when the model to code transformation, and the used code templates properly implement the functionality described in the model. It may be argued that the increased complexity of controllers that are built to support any type of request reduces security compared to simple controllers for each endpoint. However, if the endpoints were implemented individually the probability of any single endpoint controller containing flaws would have increased.

The proposed model provides a very direct mapping between the database and the REST interface. Such a mapping leads to straightforward code for translating data between database queries and the REST interface. Without such a simple mapping the complexity of writing generic request handlers would have been much larger, again increasing the risk of errors.

The model definitions produced by the model were somewhat fragile. Especially the use of property and parameter names in the validation and computation definitions could lead to problems if property names were changed. The requirement to manually specify all validation and computation logic could also lead to oversights, but realistically no better method could either have been provided in this case.

If the path parameter specification contains errors, it may lead the API to updating multiple rows, even though this is not intended to be possible. For automatically generated paths this should not occur, but when the developer may in many cases have to manually modify the path template, leading to a risk of omitting important path parameters.

Yet, overall the use of MDD should have contributed to the security of the application by providing a high level and comprehensible definition of the REST API endpoints.

4.2 Security analysis based on OWASP Top Ten

The following subsection analyzes the security of the application using OWASP API Top Ten. This set of vulnerabilities does not include every vulnerability that must be taken into consideration, but provides a framework for analyzing the most critical vulnerabilities of web APIs. In particular, it puts little emphasis on encryption and browser vulnerabilities.

The proposed model provides very limited tools for defining row-level permissions (broken object level authorization was number one in OWASP API Top Ten). However, the model still allows access to rows to be restricted based on their creator or any other column with the username.

Initial user authentication on login is provided through the existing authentication facilities. These provide only password based authentication, with no option for MFA. This problem will however be solved with the transition to OIDC, which can support MFA. This transition will also lead to changes in the session management, and will likely require support for OIDC back-channel logout to maintain support for explicit logout. Authentication will not be otherwise affected by the transition and will still be based on JWTs. The main difference will be that the JWTs will be verified based on a signature using a public key provided by the authorization server, instead of by using a HMAC and symmetric cryptography.

Another option that could be considered for client authentication is mTLS. It could conceivably be used as all users are authorized to use the API by the company. This would however require the company to operate its own CA, and distribute necessary certificates to clients and the server. Computers are often shared between multiple employees in a factory setting (for example, by machine operators working different shifts). mTLS may thus be better suited for authorizing client machines that the actual users.

When the OpenAPI definition is generated, all columns are exposed by default. This could lead to excessive data exposure. It is thus necessary to remove the parameters and body properties relating to any columns containing sensitive data. The model does not allow filtering of sensitive data based on rows, for example in configuration tables. However, the initial use case envisioned is to provide access to process data which generally does not require such filtering.

The API imposes rate limiting at the reverse proxy. This ensures that all requests are rate limited regardless of to which Node.js instance the requests are forwarded to. Rate limiting at the reverse proxy is also computationally most efficient as no requests are ever sent to the Node.js instances. The reverse proxy supports path specific rate limiting, allowing us to set a stricter limit for the login route.

Limiting the size of JSON in requests is supported by the JSON parser used for the backend. By default it sets a limit of 100 kB, which should be sufficient for our use case. Response data is not limited for single rows. This ensures that any full row in the database can always be returned. There however exists a limit on the number of rows returned. The default we have chosen is 100. If more rows are required, the can be fetched using subsequent paginated requests.

Function level authorization is specified using permissions and roles. This leaves

function level authorization as a responsibility of the maintainer of user roles and permissions, which are not part of this new backend. In order to ease management of sensitive permissions, one option is to group sensitive permissions together. For example, administrative access could be grouped together under a `api.admin.*` category to reduce the risk of accidentally assigning permissions.

Mass assignment is not possible in the proposed architecture. It exposes no endpoints which allow modification of more than one row at a time. The only case where this could happen, is if the model is incorrectly specified. If the path parameters of a row-oriented resource type do not refer to unique rows in the table as intended, assignment would be performed on all colliding rows on update and delete.

Security misconfiguration is a complicated issue, as it can occur at any level of the application stack. It is not possible to avoid all cases of misconfiguration through architectural decisions. Many software components today strive to provide secure defaults. Not changing security related settings unless required is thus desirable, especially if the personal understanding of these settings is limited. We have for example kept changes to the TLS configuration minimal by only limiting it to TLS 1.3, and use mostly defaults for CORS handling (except explicit whitelisting of domains and headers).

All database queries use prepared statements and bind parameters to prevent SQL injection attacks. The SQL queries can be pre-generated for all request handlers except PATCH handlers of row-oriented resources and GET handlers for table oriented resources. These handlers still use bind parameters for supplying user data. Bind parameters supply data directly to the database, which handles it securely without parsing it as a part of SQL statements.

The construction of SQL requests in the the GET handler for table-oriented resources and PATCH handler for row-oriented resources is not ideal even if precautions are being taken. Construction of SQL requests always carries a risk of enabling SQL injection attacks. Still, in the implementation presented above, only data included in the model can be included in SQL queries. In GET requests the only data that can dynamically alter the SQL during handler execution is the query parameter names. These affect bind parameter names, as well as the column names and operators in the WHERE clause which are derived from the model. SQL injection through the query parameters is prevented by rejecting any request containing query parameters that are not present in the model. On the other hand, in PATCH requests the dynamic portion of the PL/SQL code is based on body parameters. Which parameters are present affects dynamic validation, computation of dependent columns (including definition of PL/SQL variables for the results), as well as the row names and bind parameters in the final UPDATE statement. PATCH requests require considerably more dynamic PL/SQL that the GET requests, which increases the risk of exploitable coding mistakes. However, the only user provided data that is directly included in the PL/SQL code is the parameter names, which are protected in the same way as the query parameters in GET requests. All other information required is extracted from definitions in the model. This once again emphasizes that the main risk for SQL injection comes from the logic derived from the model, and not from user supplied

request data.

Assets management falls somewhat out of scope for this thesis, as it is mostly concerned with the implementation of an architecture for the API and not its deployment or maintenance. What still is included in the scope of the thesis is documentation and management of old versions of the API. The API is largely documented by the OpenAPI definition used as a model by the backend. As the backend functionality is directly based on the contents of the OpenAPI definition, it should always stay up to date with the current version of the API. For use as documentation the definition should be augmented with descriptions and summaries to provide a human readable description of the API functionality. Care should be taken to also document aspects such as CORS, which are not part of the model. In addition to this, documentation for development and deployment must be maintained. Any implementation details provided in the model should be stripped before sending the OpenAPI dfinition to API users. Otherwise, system internals would be exposed to third parties. One possible exception is the authorization information, as it is useful for client developers in order to understand the permission requirements of their users.

The logging implemented for the application logs security relevant events. However it does not log all data in the request, such as headers and body data. This may be relevant for forensics. Sensitive data should not be logged, as this type of data should not be included in the path or query.

4.3 HTTPS and TLS

The API employed HTTPS to protect all endpoints. The use of HTTPS was also limited to not use versions of TLS older than 1.3. In addition to this the use of the HSTS header should prevent unintentional requests using HTTP that may be intercepted. Overall the use of a recent version of TLS should provide reasonable guarantees for secure encrypted communication.

As the API is intended for use only by authorized users and services at a paper mill mTLS could have been used to further secure the access to the API. This would have required the IT department at the company to set up its own internal CA, and distribute certificates to all client machines. As computers at a mill are often shared by multiple employees (for example when working different shifts) such a setup would mainly be useful for identifying authorized machines and not users.

In addition to user authentication, (m)TLS could also have been used to protect the connections between the Node.js instances and the reverse proxy, the Redis cache and the database. Such a setup would have been particularly important if the API was to be deployed in a cloud setting.

4.4 Browser vulnerabilities

In addition to the general vulnerabilities described above, there are browser specific vulnerabilities that the API should mitigate as well as possible. These vulnerabilities are dependent on on the website using the API, and thus cannot be prevented

completely from the API. There are still a number of features available to the API that prevent some types of attacks.

CSRF attacks perform API requests from an unauthorized origin. Such attacks can be prevented by deploying CORS and rejecting any no-CORS requests as well as requests from unauthorized origin. Due to the design of the API, where authentication data is supplied through the `Authorization` header, and all body data is in JSON format, no no-CORS requests can pass validation.

The requirement of being authenticated to perform any requests, combined with the use of the `Authorization` header for authentication instead of a cookie should protect the backend against CSRF attacks, as authorization information is not available to an attacker. The backend does not employ CSRF tokens, but this can be considered unnecessary when cookie-based authentication is not used.

In addition to CORS, the backend uses the fetch metadata request headers to prevent no-CORS and same-origin requests (an API does not host web pages that make requests). Requests not originating from script based fetches are also forbidden to prevent malicious form submissions, even if these should not be possible in a JSON based API. In addition, any user originated requests were forbidden. A weakness of the fetch metadata request headers is that the specification requires requests with any unknown values for the headers to be accepted in order to promote forward compatibility. The list of forbidden values must thus be extended whenever a new value is added to the specification. It is also necessary to permit all requests omitting these headers to support non-browser clients. Older browsers may also not include these headers, but the API is intended for usage in a corporate environment where the used browsers are controlled by the IT department.

As a final protection against CSRF the CORP header is set to deny all cross-origin no-CORS requests. This header does not prevent the requests from being fulfilled, and still allows modification of the database. It only prevents the requesting website from accessing the response data. As the purpose of a CSRF attack generally is to modify the state of an application, CORP alone is not a sufficient protection against CSRF attacks, however it provides an additional defense-in-depth layer of protection in addition to the above protections.

Protection against XSSI is achieved through the same measures as for CSRF protection. The API is already not vulnerable to traditional XSSI attacks that leak data from cross-site scripts, as it only serves JSON data and no scripts. As JSON arrays can be interpreted as valid Javascript, there however still exists a possibility of XSSI attacks. CORS provides no protection against XSSI as requests originating from `<script>` tags are exempted. The handling of fetch metadata request headers prevents any requests from `<script>` tags in supported browsers, and CORP also prevents the website from reading any fetched data. Even with these protections, old browsers that support neither the fetch metadata request headers nor CORP are potentially vulnerable to XSSI.

XSS attacks can only be mitigated from an API by sanitizing user input, preventing inclusion of any script tags. A secure website should not include data fetched from APIs directly in the HTML, and this practice is discouraged by modern front end frameworks. It is still important for an API to provide this functionality, as it

provides an additional layer of protection for insecure clients.

Some of the browser based security measures must be modified if the API is to support file downloads in the future. In particular, user originated navigation requests must be allowed for any file download. These endpoints must also properly implement the `Content-Disposition` header to prevent rendering included files in the browser. If file download support was added, it would be important to limit the allowed file types. This could be included in the model by using MIME types. Additionally setting the `X-Content-Type-Options` to prevent MIME type sniffing would be particularly important for file download endpoints.

4.5 Special considerations for legacy systems

The main special considerations required for the legacy system were interoperability and proper validation of client supplied data. It may also be argued that the proposed architecture and meta-model are largely based on the specifics of the legacy system. Particularly, the implementation of computation of column values is highly dependent on the database based PL/SQL computation of values, and even includes special handling for status codes and errors.

To ensure interoperability, the options available for certain security aspects of the backend had to be restricted. Authentication was limited to the existing password based authentication, and does not support MFA (even though this will change once OIDC is deployed). Additionally, authorization was limited to mainly RBAC to preserve compatibility with the existing solutions. Still, the actual permissions used for RBAC are not interoperable with the program based solution of the legacy system.

Correct validation was a critical functionality for the new backend. Insufficient validation may have put the system in a state where assumptions of the legacy system would have been violated, leading to it malfunctioning or simply ceasing to function altogether. Validation was also the most complex aspect of the new backend, as finding the necessary validation parameters based on the existing application would have been a challenging task. The best method of verifying the sufficiency of validation may be extensive testing and consultation with maintainers of the legacy system.

4.6 Remaining security issues

Although the REST API backend fulfills most of the requirements for a secure API there are some aspects that could still be improved. Particular functionality that could be improved includes authentication and authorization. Initial authentication of users based on passwords is not the most secure option. This concern may however be alleviated by a transition to OIDC. However, OIDC introduces new security concerns such as tokens remaining valid for a time after a user has finished using the API. No conclusive answer are given when it comes to best practices for initial authentication using OIDC.

Authorization is mainly provided through RBAC. It would be desirable to introduce ABAC functionality such as authorization based on work times or location. It would also be desirable to improve the ReBAC functionality to better account for relationships between pairs of objects or subjects.

The final architecture can not be considered to adhere to the zero-trust model. It does authenticate users and validates all user supplied data. However, it does not provide MFA, and lacks authentication between some individual software components. For example, the backend does not authenticate the database.

5 Summary

The objective of this thesis was to present a model driven approach to implementing a REST API backend for a legacy system, and analyze the security of this backend. Additionally the thesis wanted to investigate if there were special considerations required to securely implement a REST API for a legacy system.

The thesis presented a meta-model for implementing a REST API for the legacy system. The meta-model was based on the OAS to provide a framework for defining the API, and augmented it with mappings for a relational database. Additionally, functionality specific to the legacy system was included in the model, most notably the concept of computations. The meta-model also included definitions for dynamic, database based validation as well as authorization, specifically RBAC and rudimentary ReBAC. A mapping between the data types of the database and OAS was also specified.

The thesis continued to discuss the partial generation of the model of the REST API from database constructs. It then continued with the implementation of the model, which did not rely on traditional code generation. It analyzed the most central implementation details of the resulting request handlers.

In addition to the model, the thesis also presented how other security features were implemented. This included headers providing additional security for browsers, as well as implementation of HTTPS, authentication and logging.

The thesis analyzed the security of the application, using OWASP API Top Ten ad common browser vulnerabilities as a framework. It concluded that the backend provided adequate security, but that certain areas of the implementation of security features could have been improved. In particular, authorization was an area where the backend could have implemented a more secure scheme, such as ABAC.

Certain considerations for the application had to be kept in mind when implementing the backend due to the legacy system. To improve interoperability, the same authentication method as in the legacy system was employed in the new backend. Additionally, the RBAC portion of the authorization framework was based on the features of the authorization framework, even though it did not follow its conventions completely. A second concern related to the legacy backend was that of proper validation of user supplied data. This was argued that this was an especially important consideration as no data that the legacy client would fail to interpret could be stored to the database.

The thesis did not use any formal framework for analysing the security of the application, neither did it include any testing of its security properties. It solely relied on observations relating to common security vulnerabilities. The thesis could have benefited from a more formal method of security analysis.

The author was not able to find any previous research that studied the use of MDD to model security aspects of a REST API, particularly when using OAS as a basis for the meta-model. The implementation of a REST API using an MDD approach was presented in [58], but it did not discuss security. Most MDS research focused on generic applications and did not take REST API specific security concerns into account.

As further research it would be interesting to generalize the provided meta-model to fit any SQL database centric legacy application. This includes support for more generic SQL queries, for example to include details from another table in the response. Another area of research is the modeling of the connection between a REST API and legacy applications that rely more on server logic than on database logic.

This thesis does not include any details on the validation of the correctness of the model, for example if computed columns are correctly marked as read only. Additional research could be targeted at providing such validation. A related subject if the validation of model generation and code generation tooling.

This thesis excludes testing of the application. The use of OAS as a basis of the meta-model lends the system to both black box testing based on the specification, and test generation using Model Driven Testing (MDT). Additionally exhaustive testing of the generic request handlers could be investigated.

References

[1] R. Fielding, Ed., M. Nottingham, Ed. and J. Reschke, Ed., "HTTP Semantics," STD 97, RFC 9110, June 2022, doi: 10.17487/RFC9110. [Online]. Available: https://www.rfc-editor.org/info/rfc9110

[2] E. Rescorla, "HTTP Over TLS," RFC 2818, May 2000, doi: 10.17487/RFC2818. [Online]. Available: https://www.rfc-editor.org/info/rfc2818

[3] L. Dusseault and J. Snell, "PATCH Method for HTTP," RFC 5789, March 2010, doi: 10.17487/RFC5789. [Online]. https://www.rfc-editor.org/info/rfc5789

[4] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[5] T. Yarygina, "RESTful Is Not Secure," in L. Batten, D. Kim, X. Zhang, G. L, (eds) Applications and Techniques in Information Security. ATIS 2017. *Communications in Computer and Information Science*, vol 719, 2017. Springer, Singapore. doi: 10.1007/978-981-10-5421-1_12.

[6] OpenAPI Initiative. "OpenAPI Specification." https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.3.md (accessed 18 Nov. 2022).

[7] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON Schema," in *Proceedings of the 25th International Conference on World Wide Web (WWW '16)*, 2016, pp. 263–273, doi: 10.1145/2872427.2883029.

[8] *Zero Trust Architecture*, SP 800-207, National Institute of Standards and Technology, USA, Aug. 2020.

[9] C. Buck, C. Olenberger, A. Schweizer, F. Völter and T. Eymann, "Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust," *Computers & Security*, vol. 110, 2021, 102436, doi: 10.1016/j.cose.2021.102436.

[10] OWASP "OWASP API Top Ten." OWASP.org. https://github.com/OWASP/API-Security/raw/master/2019/en/dist/owasp-api-security-top-10.pdf (accessed 12 Dec. 2022)

[11] T. Taya et al., "An Automated Vulnerability Assessment Approach for WebAPI that Considers Requests and Responses," *2022 24th International Conference on Advanced Communication Technology (ICACT)*, 2022, pp. 423-430, doi: 10.23919/ICACT53585.2022.9728941.

[12] A. van den Berghe, K. Yskout and W. Joosen, "A Reimagined Catalogue of Software Security Patterns," *2022 IEEE/ACM 3rd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, 2022, pp. 25-32, doi: 10.1145/3524489.3527301.

[13] M. Idris, I. Syarif and I. Winarno, "Development of Vulnerable Web Application Based on OWASP API Security Risks," 2021 International Electronics Symposium (IES), 2021, pp. 190-194, doi: 10.1109/IES53407.2021.9593934.

[14] K. Amirtahmasebi, S. R. Jalalinia and S. Khadem, "A survey of SQL injection defense mechanisms," *2009 International Conference for Internet Technology and Secured Transactions, (ICITST)*, 2009, pp. 1-8, doi: 10.1109/ICITST.2009.5402604.

[15] J. Svacina, J. Raffety, C. Woodahl, B. Stone, T. Cerny, M. Bures, D. Shin, K. Frajtak and P. Tisnovsky, "On Vulnerability and Security Log analysis: A Systematic Literature Review on Recent Trends," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems (RACS '20)*, 2020, pp. 175–180, doi: 10.1145/3400286.3418261.

[16] A. Chuvakin and G. Peterson, "Logging in the Age of Web Services," in *IEEE Security & Privacy*, vol. 7, no. 3, pp. 82-85, May-June 2009, doi: 10.1109/MSP.2009.70.

[17] A. Barth, C. Jackson and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security (CCS '08)*, 2008, pp. 75–88, doi: 10.1145/1455770.1455782.

[18] S. Lekies, B. Stock, M. Wentzel and M. Johns "The Unexpected Dangers of Dynamic JavaScript," in *Proceedings of the 24th USENIX Security Symposium*, Aug. 2015, pp. 723-735.

[19] S. Gupta, B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, iss. 1, pp. 512–530, 2017, doi: 10.1007/s13198-015-0376-0

[20] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018, doi: 10.17487/RFC8446. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[21] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280, May 2008, doi: 10.17487/RFC5280. [Online]. Available: https://www.rfc-editor.org/info/rfc5280

[22] H. Lee, D. Kim and Y. Kwon, "TLS 1.3 in Practice:How TLS 1.3 Contributes to the Internet," in *Proceedings of the Web Conference 2021 (WWW '21)*, 2021, pp. 70–79, doi: 10.1145/3442381.3450057.

[23] J. A. Berkowsky and T. Hayajneh, "Security issues with certificate authorities," 2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON), 2017, pp. 449-455, doi: 10.1109/UEMCON.2017.8249081.

[24] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP," RFC 6960, June 2013, doi: 10.17487/RFC6960. [Online]. Available: https://www.rfc-editor.org/info/rfc6960

[25] P. Hallam-Baker, "X.509v3 Transport Layer Security (TLS) Feature Extension", RFC 7633, October 2015, doi: 10.17487/RFC7633.[Online]. https://www.rfc-editor.org/info/rfc7633

[26] B. Laurie, "Certificate transparency" Communications of the ACM, vol. 57, iss. 10, pp. 40–46, October 2014, doi: 10.1145/2659897.

[27] A. Lavrenovs and F. J. R. Melón, "HTTP security headers analysis of top one million websites," *2018 10th International Conference on Cyber Conflict (CyCon)*, 2018, pp. 345-370, doi: 10.23919/CYCON.2018.8405025.

[28] A. Barth, "The Web Origin Concept," RFC 6454, Dec. 2011, doi: 10.17487/RFC6454. [Online] Available: https://www.rfc-editor.org/info/rfc6454

[29] H. Saiedian and D. Broyle, "Security Vulnerabilities in the Same-Origin Policy: Implications and Alternatives," in *Computer*, vol. 44, no. 9, pp. 29-36, Sept. 2011, doi: 10.1109/MC.2011.226.

[30] A. Barth, "HTTP State Management Mechanism," RFC 6265, Apr. 2011, doi: 10.17487/RFC6265. [Online]. https://www.rfc-editor.org/info/rfc6265

[31] WHATWG. "Fetch." WHATWG.org. https://fetch.spec.whatwg.org/ (accessed 1 Dec. 2022)

[32] J. Reschke, "Use of the Content-Disposition Header Field in the Hypertext Transfer Protocol (HTTP)," RFC 6266, June 2011, doi: 10.17487/RFC6266. [Online]. Available: https://www.rfc-editor.org/info/rfc6266

[33] M. West and A. Sartoni. "Content Security Policy Level 3." W3.org. https://www.w3.org/TR/CSP3/ (accessed 1 Dec. 2022)

[34] D. Ross and T. Gondrom, "HTTP Header Field X-Frame-Options," RFC 7034, Oct. 2013, doi: 10.17487/RFC7034. [Online]. Available: https://www.rfc-editor.org/info/rfc7034

[35] J. Hodges, C. Jackson and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, November 2012, doi: 10.17487/RFC6797. [Online]. Available: https://www.rfc-editor.org/info/rfc6797

[36] M. West. "Fetch Metadata Request Headers." W3.org. https://www.w3.org/TR/fetch-metadata/ (accessed 1 Dec. 2022)

[37] J. Reschke, "The 'Basic' HTTP Authentication Scheme," RFC 7617, September 2015, doi: 10.17487/RFC7617. [Online]. Available: https://www.rfc-editor.org/info/rfc7617

[38] M. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage," RFC 6750, October 2012, doi: 10.17487/RFC6750. [Online]. https://www.rfc-editor.org/info/rfc6750

[39] S. Ahmed and Q. Mahmood, "An authentication based scheme for applications using JSON web token," *2019 22nd International Multitopic Conference (INMIC)*, 2019, pp. 1-6, doi: 10.1109/INMIC48123.2019.9022766.

[40] "OpenID Connect." OpenID. https://openid.net/connect/

[41] D. Hardt, Ed., "The OAuth 2.0 Authorization Framework," RFC 6749, October 2012, doi: 10.17487/RFC6749. [Online]. Available: https://www.rfc-editor.org/info/rfc6749

[42] F. B. Schneider, "Least privilege and more [computer security]," in *IEEE Security & Privacy*, vol. 1, no. 5, pp. 55-59, Sept.-Oct. 2003, doi: 10.1109/MSECP.2003.1236236.

[43] R. S. Sandhu, "Role-based Access Control," *Advances in Computers*, vol. 46, pp. 237-286, 1998, doi: 10.1016/S0065-2458(08)60206-5.

[44] E. Coyne and T. R. Weil, "ABAC and RBAC: Scalable, Flexible, and Auditable Access Management," in *IT Professional*, vol. 15, no. 3, pp. 14-16, May-June 2013, doi: 10.1109/MITP.2013.37.

[45] P. W. L. Fong, "Relationship-based access control: protection model and policy language," in *Proceedings of the first ACM conference on Data and application security and privacy (CODASPY '11)*, 2011, pp 191–202, doi: 10.1145/1943513.1943539.

[46] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo and J. Voas, "Attribute-Based Access Control," in *Computer*, vol. 48, no. 2, pp. 85-88, Feb. 2015, doi: 10.1109/MC.2015.33.

[47] D. Servos and S. L. Osborn, "Current Research and Open Problems in Attribute-Based Access Control" ACM Computing Surveys vol. 49, iss. 4, art. 65, pp. 1-45, Dec. 2017, doi: 10.1145/3007204.

[48] T. Ahmed, R. Sandhu and J. Park, "Classifying and Comparing Attribute-Based and Relationship-Based Access Control," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17)*, 2017 pp. 59–70, doi: /10.1145/3029806.3029828.

[49] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Syst J*, vol. 45, (3), pp. 451-461, 2006. Available: https://www.proquest.com/scholarly-journals/model-driven-development-good-bad-ugly/docview/222423190/se-2.

[50] J. Whittle, J. Hutchinson and M. Rouncefield, "The State of Practice in Model-Driven Engineering," in *IEEE Software*, vol. 31, no. 3, pp. 79-85, May-June 2014, doi: 10.1109/MS.2013.65.

[51] G. Liebel, N. Marko, M. Tichy, A. Leitner, J. Hansson, "Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain," in J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfran, (eds) Model-Driven Engineering Languages and Systems. MODELS 2014. *Lecture Notes in Computer Science*, vol. 8767, 2014, doi: 10.1007/978-3-319-11653-2_11.

[52] P. H. Nguyen, J. Klein, Y. Le Traon and M. E. Kramer, "A Systematic Review of Model-Driven Security," 2013 20th *Asia-Pacific Software Engineering Conference (APSEC)*, 2013, pp. 432-441, doi: 10.1109/APSEC.2013.64.

[53] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security: From UML models to access control infrastructures," *ACM Trans. Softw. Eng. Methodol*, vol. 15, no. 1, pp. 39–91, 2006, doi: 10.1145/1125808.1125810

[54] C. Zolotas, T. Diamantopoulos, K. C. Chatzidimitriou, et al. "From requirements to source code: a Model-Driven Engineering approach for RESTful web services," *Automated Software Engineering*, vol. 24, pp. 791–838, 2017, doi:10.1007/s10515-016-0206-x.

[55] S. Schreier, "Modeling RESTful applications," in *Proceedings of the Second International Workshop on RESTful Design (WS-REST '11)*, 2011, pp. 15–21, doi: 10.1145/1967428.1967434.

[56] I. Koren and R. Klamma, "The Exploitation of OpenAPI Documentation for the Generation of Web Frontends," in *Companion Proceedings of the The Web Conference 2018 (WWW '18)*, 2018, pp. 781–787, doi: 10.1145/3184558.3188740.

[57] H. Ed-douibi, J. L. Cánovas Izquierdo, F. Bordeleau and J. Cabot, "WAPIml: Towards a Modeling Infrastructure for Web APIs," *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2019, pp. 748-752, doi: 10.1109/MODELS-C.2019.00116.

[58] D. Sferruzza, J. Rocheteau, C. Attiogbé, A. Lanoix, "A Model-Driven Method for Fast Building Consistent Web Services from OpenAPI-Compatible Models," in Hammoudi, S., Pires, L., Selic, B. (eds) Model-Driven Engineering and Software Development. MODELSWARD 2018. *Communications in Computer and Information Science*, vol 991, 2019, doi: 10.1007/978-3-030-11030-7_2.

[59] "Oracle Database Documentation." Oracle Help Center. https://docs.oracle.com/en/database/oracle/oracle-database/index.html