Joakim Moe Adolfsen, William Jarbeaux, Thomas Bakken Moe, Eric Younger

Digital Evidence Management

How can secure handling and storage of user-defined heterogeneous data be accomplished in a multi-tenant solution?

Bachelor's project in Computer Engineering

Supervisor: Nils Tesdal

May 2021

Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Joakim Moe Adolfsen, William Jarbeaux, Thomas Bakken Moe, Eric Younger

Digital Evidence Management

How can secure handling and storage of user-defined heterogeneous data be accomplished in a multi-tenant solution?

Bachelor's project in Computer Engineering
Supervisor: Nils Tesdal
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of Science and Technology

Preface

This project has its roots in a 2020 summer project that Thomas and Eric did for Signicat. During a conversation with Tor Even Dahl in the fall off 2020; the possibility of continuing the summer project as a bachelor project came up. Tor Even suggested a project: to take the product ”Digital Evidence Management” from a proof-of-concept to a product ready to be deployed in a demo environment. To be part of a project from inception to production code seemed like a unique opportunity. This was one of the main draws for the group when choosing this project.

During the project, we've been working in close proximity with Signicat. When the pandemic has allowed for it: three members of the team members worked at Signicat's offices in Trondheim while one worked remote from Oslo. The project followed a SCRUM structure with the management of ceremonies such as stand-ups, retrospectives, and reviews left to the team. The project started with exploration of key technologies, then moved over to development of the main product, and later on to deployment and ”DevOps”-work.

We extend our thanks and sincere appreciation to Tor Even Dahl, our Product Manager from Signicat. He has played a monumental role in getting the Digital Evidence Management (DEM) project started and adapting it into a bachelor project. As the Product Manager, he has helped steer the project in the right direction and guided us when tackling design questions. Tor is a real firebrand and a driving force behind the project. DEM, and thus our bachelor project, would not exist without him.

Other people instrumental to the project are Ansgar Tasler, Dag Sneggen and Steinar Knutsen from Signicat. Ansgar Tasler has been a great advisor in terms of Dev-ops when we were deploying our system to AWS. He has greatly helped us to understand more on how the deployment process works, and was a key person in making the deployment easier.

Dag Sneggen has been a great resource to us in regards to both understanding the OAuth/OIDC protocol and in terms of giving us the knowledge we needed in order to secure the Digital Evidence Management system.

Steinar Knutsen has been a supporting figure for our team through advice and guidance under the development process.

We would also like to thank Nils Tesdal, our supervisor from NTNU. He has helped to guide us throughout the project.

Finally we would like to say thanks to our classmates, who have been a source of great help, advice and motivation during the course of this project and the three years of studies preceding it.

Assignment

The assignment given to us was to develop a new system for managing sensitive data such as consent forms and transaction records. The main portion of this system was to be an API that timestamp, stores and verifies incoming records. We were also tasked with creating a demo application to showcase the usages of the API and serve as an example for potential customers. The assignment specified the use of several technologies such as OIDC and trusted timestamping. Another requirement was that the system would have to use multi-tenancy, but could support being used in a single-tenant solution

Signicat laid out many functional and non-functional requirements for DEM. These requirements can be viewed in full in attachment 8.3 or attachment 8.4. An overview of which requirements we hit or not can be found in chapter 4.

The original goal of the assignment was to complete and deploy a production-ready system by the end of the project. During development and after some deliberation between the developer team and the product owner/manager, it was determined that this would not be achievable within this the timeframe of the project. Instead the scope was changed to create a hosted demo version of the system which could then be presented to potential customers. This allowed us to focus more on the system itself, and less on all the requirements of a production-ready application.

Abstract

The amount of digital information that is being tracked and stored by companies is increasing every year. Many companies collect this information, but do not have any dedicated systems to manage it appropriately. In recent times several regulations have been introduced that force companies to keep better control of the data in their possession.

This report details the work of creating an application for managing sensitive data such as consent forms, transaction records, etc. The work resulted in a easy to use API and an example of its use case in the form a web application.

The main challenges of the assignment were secure handling of data, and being able to process data that is largely defined by the user (heterogeneous data). We have also had to consider different ways of storing the data and weigh what is ideal up against what is realistic for a real product.

The project has been developed in collaboration with Signicat AS, and has adhered to many of their standards and technical requirements. The main work methodology used was Scrum with involvement from Signicat throughout the development.

Contents

1 Introduction **10**

1.1 Terms 11

1.2 Acronyms 11

2 Theory **13**

2.1 Design patterns 13

2.1.1 Separation of concerns 13

2.1.2 Data Access Object (DAO) 13

2.1.3 Model-View-Controller (MVC) 13

2.1.4 Facade 13

2.2 HTTPS 14

2.3 OAuth 2.0 14

2.3.1 OIDC 14

2.3.2 Client Credentials Flow 15

2.3.3 Authorization Code Flow with PKCE 16

2.4 PSD2 18

2.5 Scrum 18

2.5.1 Sprint planning and Daily Scrum 19

2.5.2 Sprint Review and Retrospective 19

2.5.3 Product Backlog 19

2.5.4 Sprint Backlog 19

2.5.5 Increment 19

2.6 Single & Multi-tenancy 20

2.7 REST 20

2.8 Relational and non-relational databases 21

2.9 ACID 21

2.10 TSA / TSP 22

3 Technology and methods **23**

3.1 RQ Requirement 1: Authenticate the user 23

3.1.1 Signicat Express 23

3.1.2 OAuth 2.0 / OIDC 23

3.2 RQ Requirement 2: Contain the heterogeneous data 23

3.2.1 MongoDB 24

3.3 RQ Requirement 3: Manage ownership of data in the database 25

3.4 RQ Requirement 4: Keep the data secure in transit and at rest 25

3.5 RQ Requirement 5: Verify that the data has not been modified 25

3.6 API 26

3.6.1 Spring Security 26

3.6.2 Java 26

3.6.3 Maven 26

3.6.4 JUnit 27

3.7 Demo Application 27

3.7.1 TypeScript 27

3.7.2 React 28

3.7.3 Formatting Tools 28

3.7.4 Figma 28

3.8 Common technologies and tools 29

3.8.1 GitLab 29
3.8.2 Docker 29
3.8.3 Testing 30
3.9 Work Method 30
3.9.1 30
3.9.2 Work Management Tools 30
3.9.3 Responsibilities within the team 30

4 Results **31**

4.1 Scientific Results 31
4.1.1 Requirements for Research Question 31
4.1.2 Benchmarking 32
4.2 Engineering results 34
4.2.1 API 34
4.2.2 Demo Application 39
4.3 Deployment 42
4.4 Encryption in transit 42
4.5 GDPR Compliance 42
4.6 Merge to master 42
4.7 Snyk 43
4.8 Missing requirements 43
4.9 Administrative Results 44
4.9.1 44
4.9.2 Work distribution 45

5 Discussion **46**

5.1 Scientific Discussion 46
5.1.1 Requirements for Research Question 46
5.1.2 Benchmarking 46
5.2 Engineering Discussion 47
5.2.1 Development Tools 47
5.2.2 Query 47
5.2.3 Deployment 48
5.2.4 Encryption in transit 49
5.2.5 Encryption at rest 49
5.2.6 OIDC Flow 50
5.2.7 Other requirements not met 51
5.3 Administrative Discussion 53
5.3.1 53
5.3.2 Teamwork 53
5.4 Societal Perspective 55
5.5 Professional Ethics 55

6 Conclusion **57**
6.1 Further work 58

7 References **59**
7.1 Personal Communication 62

8 Attachments **63**
8.1 System Documentation 64
8.2 Benchmarking Results 122

8.3 Requirement Documentation 131

8.4 Vision Documentation 144

8.5 Process Document 159

List of Figures

1 Example of how extracted claims from an ID token can look like [28]. 15

2 Client credentials flow [8] 15

3 Example of an access token response using the client-credentials flow 16

4 Visual example of the PKCE flow [6]. 17

5 The difference between single-tenant and multi-tenant solution [40] 20

6 Timestamp request [23]. 22

7 Timestamp response [23]. 22

8 Record, the data structure that gets stored in the database. 24

9 Time distribution per team member 45

1 Introduction

The origin of this assignment lies in Signicat's desire to develop a new system for managing sensitive data such as consent forms, transaction records, or any number of individually defined records. The idea came about when Signicat communicated with its banking partners about their need for a system to meet the requirements of the EU Directive Payment Services Directive 2 (PSD2), in particular the requirement of dynamic linking. Though this was the initial spark of the concept, Signicat wanted to develop something that can be used in a wide variety of use cases.

Tor Even Dahl from Signicat had this to say about Signicat's motivation for the project:

> ”The initial motivation for making Evidence Management was the need from regulated businesses where they need to safely store audit events and [have a] fast search and retrieve on such [events]. It was mainly about fulfilling requirements from regulations and country finance authorities defining strict rules and external audits. Initial focus was on audits on payment transactions. Customers also recognised other needs for a common event audit for GDPR and other user activities.”
> - Tor Even Dahl, Slack message, 19.05.2021

Based on the assignment presented to us by Signicat, we formulated the following research question:

”How can secure handling and storage of user-defined heterogeneous data be accomplished in a multi-tenant solution?”

We have identified the following requirements to satisfy this research question:

1. Authenticate the user.
2. Contain the heterogeneous data.
3. Manage ownership of data in the database.
4. Keep the data secure in transit and at rest.
5. Verify that the data has not been modified.

1.1 Terms

heterogeneous data

Data with a high variability of data types and formats. [44]. 3, 23, 24, 33, 37

1.2 Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 5, 21, 25

API Application Programming Interface. 2, 3, 5, 23–26, 29, 34–38, 40, 42, 43, 45, 47, 50–53, 58

AWS Amazon Web Services. 1, 25, 29, 40, 42, 48, 49

CD Continuous Deployment. 29, 48, 49

CI Continuous Integration. 29, 35, 39, 42, 47–49

CSS Cascading Style Sheet. 39

DAO Data Access Object. 4, 13, 37

DEM Digital Evidence Management. 1, 2, 18, 23, 25, 26, 36, 47, 50, 51, 53, 55, 57

eIDAS Electronic Identification, Authentication, and Trust Services. 22

EU European Union. 18, 22

GDPR General Data Protection Regulation. 10, 42

GUI Graphical User Interface. 29, 40

HATEOAS Hypermedia as the Engine of Application State. 52

HTML Hypertext Markup Language. 39

HTTP Hypertext Transfer Protocol. 14, 32, 33, 36, 37

HTTPS Hypertext Transfer Protocol Secure. 4, 14, 25, 42, 49

JSON JavaScript Object Notation. 24, 34, 35

M2M Machine-to-machine. 15

MVC Model-View-Controller. 4, 13, 36

NoSQL non-relational. 32

NTNU Norwegian University of Science and Technology. 1, 53

OAS OpenAPI Specification. 38

OIDC OpenID Connect. 1, 2, 4, 5, 14, 23, 25, 37, 47

PKCE Proof Key for Code Exchange. 4, 9, 16–18

POC Proof Of Concept. 26

POJO Plain Old Java Object. 52

PSD2 Payment Services Directive 2. 4, 10, 18

QTSA Qualified Time-Stamping Authority. 22, 25, 26

QTSP Qualified Trust Service Provider. 22

RDBMS Relational Database Management System. 21

REST Representational State Transfer. 5, 20, 30, 34, 36

RFC Request for Comments. 14, 16, 18, 22

RQ Research Question. 5, 23, 25

SoC Separation of Concerns. 13, 20

SPA Single Page Application. 16, 50

SQL Structured Query Language. 21, 25, 32

TLS Transport Layer Security. 14, 25

TSA Time-Stamping Authority. 22, 25, 35, 47

TSP Time Stamp Protocol. 22

TTL Time to Live. 35, 36

URI Uniform Resource Identifier. 20

UUID Universally Unique Identifier. 25

XSS Cross Site Scripting. 50

2 Theory

This section provides information on key terms and concepts relevant to this document / project.

2.1 Design patterns

Design patterns are general solutions that are reusable and serves to solve commonly occurring problems when developing computer software [41].

2.1.1 Separation of concerns

Separation of Concerns (SoC) is a fundamental design pattern/principle. It states that an application should be separated into distinct section with their own concerns [38].

2.1.2 Data Access Object (DAO)

Data Access Object (DAO) is a software design pattern which isolates the application layer from the persistence layer. The DAO provides an abstract interface to any form of database, providing data operation to application layer without disclosing details about the database itself[10].

2.1.3 Model-View-Controller (MVC)

Model-View-Controller (MVC) is a design pattern that builds on the Separation of Concerns pattern. The MVC pattern divides an application into three parts: Model, View and Controller.

The model component is typically some representation of real-world concepts, and often holds data of some kind. The view component handles all the functions that directly interact with the user. The controller component is the leading component that received user data and decided what to do with it[31].

2.1.4 Facade

The facade design pattern is a object or a class that provide a layer of abstraction to hide complex underlying code [20].

2.2 HTTPS

Hypertext Transfer Protocol Secure (HTTPS), also called HTTP over TLS, is the standard for in-transit security between a web browser and a website. HTTPS prevents websites from broadcasting their information in a way easily accessible way.

To keep communications secure it is encrypted by a protocol called Transport Layer Security (TLS) formerly known as SSL [47]. TLS works by having the client use the server's certificate to initiate a handshake and set up session tokens for encrypted communications [25].

2.3 OAuth 2.0

Request for Comments (RFC) 6749 explains: "The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf [30]."

The OAuth 2.0 framework uses tokens that functions as credentials. The different tokens are explained as:

- **Access token** are credentials used to access protected resources. The access token provides an abstraction layer, replacing different authorization constructs (e.g, username and password) with a single token understood by the resource server. Token represent specific scopes and duration of access, granted by the resource owner, and enforced by the resource server and the authorization server [30].
- **Refresh token** are credentials issued to the client by the authorization server and are used to obtain a new access token when the current token becomes invalid or expires [30].

2.3.1 OIDC

OpenID Connect (OIDC) is an authentication protocol that functions as an identity layer that works on top of the OAuth 2.0 Protocol. The OIDC protocol also uses ID tokens that contain claims about the authentication of an end-user by the authorization server. These claims can be extracted so that it possible to identify and verify the end user [33].

```
{
 "iss": "http://server.example.com",
 "sub": "248289761001",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "name": "Jane Doe",
 "given_name": "Jane",
 "family_name": "Doe",
 "gender": "female",
 "birthdate": "0000-10-31",
 "email": "janedoe@example.com",
 "picture": "http://example.com/janedoe/me.jpg"
}
```


Figure 1: Example of how extracted claims from an ID token can look like [28].

2.3.2 Client Credentials Flow

The client credential flow is typically used for Machine-to-machine (M2M) applications, and is different from other authentication flows by authenticating an app rather than a user. This is useful for tools and services like daemons, command-line interfaces, and services running on a back-end server [8].

The client credentials consists of a client id and a client secret, and is known as an authorization grant type in the OAuth 2.0 framework [30].

The client credentials protocol flow is shown in Figure 2.

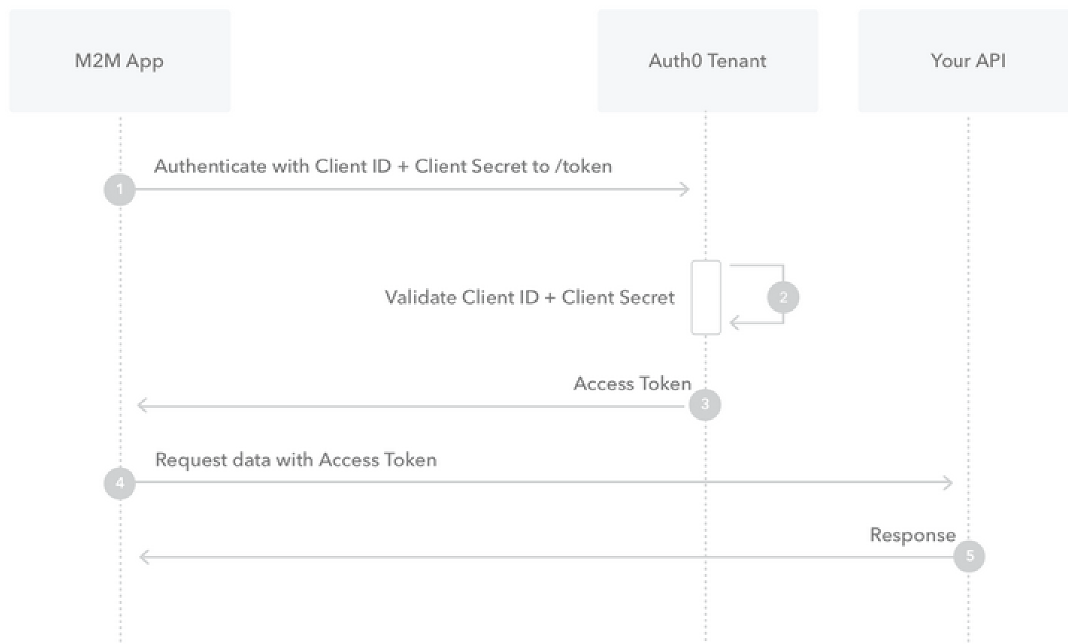


Figure 2: Client credentials flow [8]

1. An application sends in their authorization grant (client credentials) to the authorization server.
2. The authorization server verifies the authorization grant.
3. If the authorization grant is valid the authorization server returns an access token.
4. The user provides the access token when making requests to the resource server.
5. The resource server verifies the access token and returns the requested resource if the access token is valid.

On step 3, if the authorization grant is valid, then authorization server will return an access token as shown in Figure 3.


```
{
    "access_token": "eyJhbGciOiJSUzI1",
    "expires_in": 300,
    "token_type": "Bearer",
    "scope": "api://3039afb8408b4da6aa330cf536a1fa96/DEM:read "
}
```


Figure 3: Example of an access token response using the client-credentials flow

As long as the access token has not expired, the machine can provide the access token along with an request to get the requested resources. When the access token expires, the machine would have to repeat step 1-5 again.

The protocol flow explained in further detail can be read in RFC 6749 OAuth 2.0 Framework, section 1.2 [30].

2.3.3 Authorization Code Flow with PKCE

The Proof Key for Code Exchange (PKCE) flow tries to mitigate the security concerns that public clients like native applications and Single Page Application (SPA)'s pose, as they can not securely store a client secret. The PKCE authorization flow is intended for exactly this use case, and is generally considered best practice [6]. Figure 4 shows a detailed breakdown of the flow.

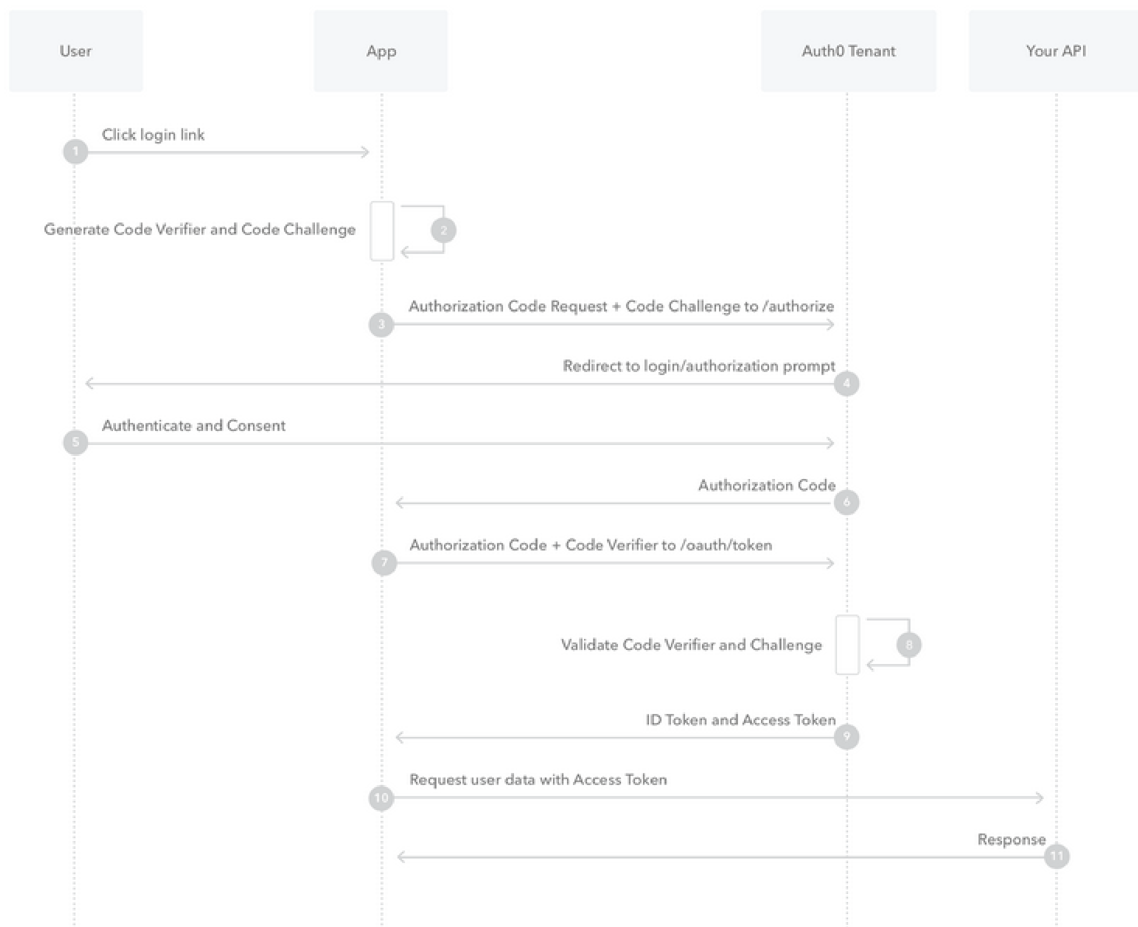


Figure 4: Visual example of the PKCE flow [6].

1. User clicks on login link on a web application.
2. The web application generates a code verifier and a code challenge.
3. The web application redirects the user along with the code challenge to a /authorize endpoint on authorization server.
4. The authorization server redirects user to login page with authorization prompt.
5. User authenticates and consents the request by clicking on a grant access button.
6. The authorization server stores the code challenge and redirects the user back to the original web application along with an authorization code.
7. The web application sends the received authorization code and the code verifier to a /oauth/token endpoint on authorization server.
8. The authorization server verifies the code challenge and the code verifier.

9. If request was valid, the authorization server responds with a access token and a ID token, and optionally a refresh token.
10. The web application sends the access token along with any request for resources to the resource server.
11. If the access token is valid, the requested resource is returned.

A more detailed explanation of the PKCE protocol can be found in the RFC 7636 document [32].

2.4 PSD2

Payment Services Directive 2 (PSD2) is an EU Directive that aims to provide a legal foundation for electronic payments within the EU. This directive sets strict security requirements, enforces transparency, and grants the users certain rights and obligations [36]. Article 21 is especially relevant to the DEM project:

"Member States shall require payment institutions to keep all appropriate records for the purpose of this Title for at least 5 years, without prejudice to Directive (EU) 2015/849 or other relevant Union law" [4].

Article 5 in the directive state that transactions should be dynamically linked to an amount and a payee specified by the payer when the transaction is initiated. Which is a requirement that can be accomplished with a system like DEM [5].

2.5 Scrum

Scrum is an agile development framework often employed in software development.

The original creators of the concept: Ken Schwaber and Jeff Sutherland, defines it as: *"Scrum is a lightweight framework that helps people, teams, and organizations generate value through adaptive solutions for complex problems"*[43].

The scrum methodology consists of 5 events: the sprint, sprint planning, daily scrum, sprint review, and sprint retrospective. During development, three artifacts are produced: the product backlog, sprint backlog, and a product increment [43].

2.5.0.1 Roles

The fundamental roles in a scrum process are the developers, a product owner and a scrum master. The developers are the people in charge of working towards a usable increment at the end of a

sprint. The product owner is accountable for the product backlog and delegating responsibilities. The scrum master is responsible for correct application of the scrum methodology and facilitating efficient work from the development team.

2.5.1 Sprint planning and Daily Scrum

A sprint is initiated by the sprint planning event. This is where the goals of the sprint are determined and that maps against the final product goal. Once the sprint has started daily scrum are conducted each day. These are at max 15-min sessions where the team inspects progress towards the sprint goal [43].

2.5.2 Sprint Review and Retrospective

Sprint Review is the first ceremony after the sprint finishes. The goal is to learn from the past sprint, inspect the outcome and determine future improvements. The Retrospective is a learning step to learn what works and should be carried over to the next sprint [43].

2.5.3 Product Backlog

The product backlog is a set of issues/tasks that the scrum team needs to do in the development of a product. There is a regular session attached to the product backlog called ”Backlog grooming” where the product owner and the team discusses the backlog items and reviews them. This makes it easier after each sprint, to choose updated backlog items to put in the next sprint [43].

From the Product backlog, the team and product owner chooses which issues to focus on on the next sprint based on priority and wishes from the product owner. It is mainly the product owners responsibility to set the content, availability, and priorities for the product backlog [37].

2.5.4 Sprint Backlog

The sprint backlog consists of a sprint goal and the set of items from the product backlog selected for this sprint. This is used primarily by the developers to visualize what must be done to achieve the sprint goal [43].

2.5.5 Increment

Increments are stepping stones towards the product goal. An increment must have a Definition of done, a formal description of it’s current state [43].

2.6 Single & Multi-tenancy

Single-tenancy and multi-tenancy describes how many users are served by the same instance of software and its supporting infrastructure. In single-tenant architecture a single user is served their own instance, while multi-tenant architecture serves multiple users on the same instance. While the data from multiple users is contained in the same place, it is still isolated and kept invisible from each other [7].

The benefits of a single-tenant architecture are those of security, dependability and customization. The main drawbacks is that of cost and maintenance. Multi-tenant architecture have the benefit of lower costs, scalability and easier configuration for the developer. [7].

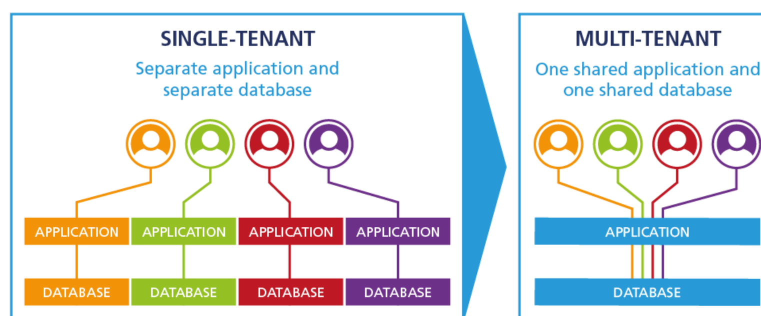


Figure 5: The difference between single-tenant and multi-tenant solution [40]

2.7 REST

Representational State Transfer (REST) is a software architectural style introduced by Roy Fielding in 2000. His dissertation defines six constraints which will make a web service "RESTful": *"The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture"* [17].

Uniform Interface. A fundamental constraint is that any interaction with a RESTful system should be uniform, as defined by four properties:

- Resource-Based
- Manipulation of Resources Through Representations
- Self-descriptive messages
- Hypermedia as the Engine of Application State (HATEOAS)

Client-server architecture. The client and the server should not be dependent on each other. This is to match with the principle of Separation of Concerns. This allows the components to evolve independently as the client communicates with the server only through resource URI's.

Stateless. All client-server interactions must be stateless. The server should not store anything

about the requests made, so that all requests will have to contain all information necessary to understand the request.

Cacheable. Data served in a response is marked as cacheable or non-cacheable. Client-side caching reduces latency and load on the server. The server can specify how long and in what way the client is allowed to cache the response.

Layered System. Layered system architecture can be deployed where the client does not know if it is connected to the end server or an intermediary.

Code on demand. The only optional constraint included. This constraint allows the server to send executable code for the client to execute.

2.8 Relational and non-relational databases

A major difference in databases are whether they are relational or non-relational. A relational database or Relational Database Management System (RDBMS) stores data in tables and rows and often use Structured Query Language (SQL) to write and query the database [45]. Relational databases have been around since the 1970s and are still popular today.

Non-relational databases, often called NoSQL databases, instead store their data in a non-tabular form. They tend to be more flexible then SQL-based structures[46]. There are in general four types of NoSQL databases: document stores, column oriented stores, key-value stores and graph stores.

The two types of databases both have pros and cons attached to them and are suitable for different types of projects. Relational databases often excel at data integrity and security while non-relational databases provide greater scalability and flexibility and are not as constrained by structure limitations. [34]

2.9 ACID

ACID refers to a set of properties that guarantee data integrity when dealing with database transactions. These properties are explained as follows [1]:

Atomicity. An entire sequence of actions must be either completed or aborted. The transaction cannot be partially successful.
Consistency. A transaction takes the resources from one consistent state to another.
Isolation. A transaction's effect is not visible to other transactions until the transaction is committed.
Durability. Changes made by the committed transaction are permanent and must survive system

failure.

2.10 TSA / TSP

RFC 3161 describes a time-stamping service as a service that supports assertions of proof that a datum existed before a particular time [23].

This is done through the use of the Time Stamp Protocol (TSP) also detailed in RFC 3161. The timestamp request is done by sending a framed byte array of the data to be timestamped, where each bit in the frame corresponds to options or parameters in the timestamping service.

A timestamp request will have the format shown in Figure 6. The timestamping service would then return a response with the format shown in Figure 7.

```
TimeStampReq ::= SEQUENCE  {
 version                      INTEGER  { v1(1) },
 messageImprint               MessageImprint,
   --a hash algorithm OID and the hash value of the data to be
   --time-stamped
 reqPolicy             TSAPolicyId              OPTIONAL,
 nonce                 INTEGER                  OPTIONAL,
 certReq               BOOLEAN                  DEFAULT FALSE,
 extensions            [0] IMPLICIT Extensions  OPTIONAL
}
```

Figure 6: Timestamp request [23].

```
TimeStampResp ::= SEQUENCE  {
  status                  PKIStatusInfo,
  timeStampToken          TimeStampToken     OPTIONAL  }
```

Figure 7: Timestamp response [23].

TSP is implemented by a Time-Stamping Authority (TSA), which serves a trusted third party in the operation. Though there are several providers of such services, the EU created the Electronic Identification, Authentication, and Trust Services (eIDAS) regulation which provides a legal framework for such services in the European Single Market. The EU keeps a list of trusted providers which can be certified as a Qualified Trust Service Provider (QTSP) [24].

A TSA provided by a QTSP are know as a Qualified Time-Stamping Authority (QTSA). Timestamps provided by a QTSA provides full data integrity to the extent where it can not be disputed in a court of law. Where as timestamps provided by a TSA can not offer the same assurances [24].

3 Technology and methods

In this section we will discuss our choices of technologies and methods, and how some of these relate to the Research Question (RQ) requirements we identified in chapter 1.

3.1 RQ Requirement 1: Authenticate the user

Before the user can access the Digital Evidence Management (DEM) system, the user needs to be authenticated. For user authentication we use Signicat Express and Signicat's own solution. As DEM is going to be a product offered by Signicat, it was only natural to incorporate DEM into Signicat Express instead of setting up a separate authentication and user management system.

3.1.1 Signicat Express

Signicat Express is a system for accessing many of Signicat's products and services. Express handles and stores the information that is required for a customer to access the products that fall under the Signicat Express umbrella. This includes client IDs, client secrets, as well as access scopes. The access scopes decide what products a customer of Signicat Express will have access to, as well as what kind of access the customer has to the products (for example: read/write) [39].

3.1.2 OAuth 2.0 / OIDC

Signicat supplies us with a intermediate authorization server from Signicat Express to handle the validation and verification of authorization grants and handles the distribution of access tokens.

On DEM's API, we only need to validate and verify access tokens, and decode the tokens using the OIDC layer. With the implementation of the OIDC layer, we can secure the resources on the API as well as integrate user isolation based on client credentials.

3.2 RQ Requirement 2: Contain the heterogeneous data

In the API we represent the heterogeneous data as Java record objects. The formatting of the Record object is the same as the data that is stored in the database (see Figure 8).

```
public record Record(
        @Id
        @JsonProperty("uuid") UUID uuid,
        @JsonProperty("type") RecordTypeEnum type,
        @JsonProperty("systemMeta") Map<String, Object> systemMeta,
        @JsonProperty("customerMeta") Map<String, Object> customerMeta,
        @JsonProperty("coreData") Map<String, Object> coreData)
        implements Serializable {
}
```

Figure 8: Record, the data structure that gets stored in the database.

Only the fields *"customerMeta"* and *"coreData"* are provided by the end-user. We represent these fields with the data type Map<String, Object>. This allows us to represent and contain any data that is delivered to the API as a JSON body. This data type also maps really easily to our chosen database type, MongoDB. MongoDB is a NoSQL database and thus especially good at handling heterogeneous data [34].

A more detailed explanation on how the Record class works can be found in attachment 8.1. System Documentation.

3.2.1 MongoDB

MongoDB is a non-relational, document-oriented database. MongoDB uses a source-available license, which means that while the source code for the database is available, the Enterprise version is not considered open source.

"MongoDB is a general purpose, document-based, distributed database built for modern application developers and for the cloud era."[26]

In the beginning of the project, we conducted several benchmarking tests. One of these tests was to compare the speed of several types of requests to different database types. We tested MySQL, PostgresSQL, and MongoDB. We also planned to test Firestore, but this fell through. The benchmarking results are attached (8.2). One of the major points we gathered from the benchmarking is that SQL databases do not perform well when trying to query unindexed heterogeneous data. Heterogeneous data is by it's nature hard to index. The poor performance of the SQL databases when querying, combined with NoSQL databases' inherent aptitude have with heterogeneous data helped us to decide on MongoDB as our database solution[34].

This choice was based on our benchmarking results and discussions with our product manager and technical lead. A combination of performance, document structure, and isolation capabilities led us to use MongoDB. Another important factor was the fact that MongoDB, as of version 4.0, supports

ACID transactions, combining the speed of a document model with guaranteed data integrity. [2]

3.3 RQ Requirement 3: Manage ownership of data in the database

We handle multi-tenancy by dividing the data in our database into separate collections (equivalent to a table in an SQL database) for each user. Each collection is named after the client's ID, provided by Signicat Express. When a user wants to access their data, we use OIDC to parse out the client ID from their provided access token. This allows us to connect the user to the correct collection. OIDC was one of the non-functional requirements laid out by our client at the beginning of the project.

3.4 RQ Requirement 4: Keep the data secure in transit and at rest

The type of data that DEM is designed to process is highly sensitive. This means that it is critical to keep this data secure during transmission (to and from the API) and at rest (stored in the database). For transmission, we use HTTPS with a TLS certificate provided by our hosting solution, Amazon Web Services (AWS). To keep the data encrypted at rest we will rely on MongoDB to handle this for us, as this is a feature in MongoDB Enterprise [14].

HTTPS was an obvious choice as it is the trusted industry standard and does not require much effort on our part as it is provided by our hosting service. The same goes for MongoDB Enterprise; we can develop quickly with MongoDB Community and upgrade to Enterprise later when we go into production.

3.5 RQ Requirement 5: Verify that the data has not been modified

To verify that the user's data has not been tampered with, we are using Signicat's QTSA service.

When we are storing a record, the core data gets timestamped by the QTSA before being saved to the database. When we retrieve a record by UUID, we hash the core data using the same hash method as the QTSA. We then compare the hashes to see if they match. If the hashes match: then the core data has not been tampered with.

By using a QTSA instead of a normal TSA, we gain the trust that the 'Q' represents. We can be absolutely sure that a record was created when the says it was and that it has not been tampered with. We can even use this as evidence in court [24].

To use Signicat's QTSA was another requirement laid out by our client. In practise, we used a TSA meant for work and pre-production (owned by signicat) during this project. This is because it is costly and considered fraud to timestamp "false" documents. When DEM goes to production, we

will switch over to using Signicat's QTSA.

3.6 API

In this section we will discuss the technologies that are specific to the API, but that do not directly apply to one of the research question requirements.

3.6.1 Spring Security

"Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications[42]".

3.6.2 Java

The DEM project started out as a POC project in the summer of 2020. The POC was written in the programming language Kotlin. In the weeks before the beginning of the bachelor project, there was an active discussion if the API in bachelor project also was to be written Kotlin. In the end, Signicat decided that the API should be written in Java, but in the newest version available. Using the newest version of Java gives some of the benefits that writing in Kotlin would give. The primary benefit is that we can use records, Java's new immutable data object type (similar to Kotlin's data classes). The API was written in Java 15 with preview enabled.

> "Java is an established language in Signicat, while Kotlin does not have the same strategic support. Kotlin was evaluated, and as a direct competitor to Java, insufficient support was given to make a move to Kotlin as a first class language in the company. The number of key features from Kotlin making their way into Java also factored into this decision."
> - Steinar Knutsen, Slack message, 08.03.21

Java 16 was released on the 16th of March, in the middle of our project. Had Java 16 been available at the beginning of the project, it would have been used. The DEM API will be upgraded to Java 16, but it was felt that the upgrade would waste precious time during the project.

3.6.3 Maven

Maven is a software management and build automation tool focused on how a project is built and manages it's dependencies[19]. Its primary function is to simplify the build process and create a

uniform build system. When choosing a build automation tool there are several options too choose from, the most popular being Maven and Gradle. We chose Maven as it was a good fit for this project as the application itself will not be overly complex. Another consideration was the fact that the client primarily use Maven in their projects and using this technology would make the hand-off process more smooth.

If the project were to grow considerably one could consider migrating over to Gradle which is a newer tool better suited for more complex applications. Gradle was designed to address the drawbacks of Maven, such as shorter build times and being more flexible[3].

3.6.4 JUnit

JUnit is a testing tool framework built for Java and the JVM. In our project, we use JUnit 5 in addition to RESTEasy for solving our testing needs. JUnit 5 is used for unit tests, while RESTEASY handles integration testing.

3.7 Demo Application

In this section we will discuss the technologies that are specific to the Demo Application, but that do not directly apply to one of the research question requirements.

3.7.1 TypeScript

TypeScript is a strict typing superset of JavaScript. The primary purpose is to enforce strict typing at compile level to make it challenging to assign non-predefined variable-types. Microsoft developed TypeScript and released it in 2012. When we considered our demo frontend application technologies, we realized that we needed to optimize the eventual production code. The next logical step was using TypeScript as our base language because it allowed our compiled JavaScript to be safer and result in a more stable application. TypeScript, in conjunction with React, makes our components only take in the types of variables that are already pre-defined in the type declaration, placed at the top of our component file. When we expand the component-files to include more sub-components, the same types could be used and made sure that there was a clear pattern to the component design. TypeScript will follow the variable path and ensure that none of our variables have changed their type throughout. The check occurs on the compilation of the application directly in our IDE, making it easy to detect any possible issues. The TypeScript compilator will not allow compilation if there are unspecified types in our code.

3.7.2 React

”React is a JavaScript library for building user interfaces [16]”. React uses a Virtual Document Object Model (DOM), which means that instead of re-rendering the entire DOM on every change to the DOM, it will only re-render the affected components that are changed on state or prop changes. This is valuable in terms of efficiency and server pressure. Since React is component-based, we can make components re-usable, instead of writing them again or having duplicate lines of code several places. React is developed by Facebook, and has a large community, and since they released React-hooks, state handling is a breeze. Considering all the aforementioned reasons and that every team member had prior experience with it, we decided that React was a good fit for our project specification and would suit our needs.

3.7.3 Formatting Tools

Eslint is a tool for identifying and reporting patterns found in ECMAScript/Javascript code, with the goal of making code more consistent and avoiding bugs [15].

Eslint comes by default with both linting and to some degree, style formatting. For our project we disabled the style formatting in Eslint and only used Eslint for linting errors. We used Prettier instead to handle the responsibility of style format enforcing, as Prettier is a stronger tool to use for that purpose.

Prettier is an opinionated code formatter, meaning that we can define our own rule-set for how we want the coding style for the project to be, and the editor will automatically format the code to follow those rules on save. This is used to increase readability throughout the project and have code style consistency. Another feature with Prettier, is that it provides auto-formatting and feedback while developing [35].

3.7.4 Figma

Figma is a visualizing tool that can be used for designing both high-end and low-end prototypes efficiently. The advantage of using Figma as a prototyping tool is exporting the prototype components to CSS. In conjunction with the fast interface, this feature enabled us to create accurate prototypes quickly. These features together made the design process precise and effective, and made Figma a natural choice as a wireframe tool [18].

3.8 Common technologies and tools

In this section we will discuss the technologies that are common for both the API and the Demo Application, but that do not directly apply to one of the research question requirements.

3.8.1 GitLab

GitLab is an online Git repository with DevOps features like Webhooks, CI, CD, container registry and Kubernetes Cluster management. We use GitLab as a central remote repository for version control, and use GitLab CI to run a build, test, deployment and publish stages.

3.8.2 Docker

Docker is platform for shipping and running applications through the use of OS-level virtualization. This is done by creating Docker images which are then instantiated in the form of Docker containers [12].

The usage of Docker was one of the requirements given to us by the client. During development we use Docker in conjunction with the tool docker-compose to create instances of the various databases, such as for benchmarking, local environment and testing. We also use this for running the Prometheus server.

We also use Docker to make images of the API and the Demo Application and thereby ”dockerize” the projects. These images are then used during the deployment to AWS. By creating Docker images of the projects, we introduced the feature of scalability. If the API for instance is under high pressure, tools like Nomad or Kubernetes can run another instance of the image and thereby alleviate the pressure by distributing the workload.

3.8.2.1 Docker-Compose

”Docker-compose is a tool for defining and running multi-container Docker applications [13]”.

This tool was especially useful when we were bench-marking multiple databases, and using GUI tools like ”phpMyAdmin” and ”pgAdmin”. We defined the databases along with any preliminary configuration in a docker-compose.yml file, and could then easily run all Docker containers with the single command ”docker-compose up”. The advantage of running databases in containers instead of as an service on each computer, is that every computer would have the same clean setup on each run.

3.8.3 Testing

For integration tests we elected to use the framework REST Assured. This allowed to test several layers of our application with a simple and intuitive framework. For unit tests we used the mocking framework Mockito which allowed us to isolate the tests to just the class in question. For the Demo Application we focused on component testing with the use of the React Testing Library and Jest.

3.9 Work Method

3.9.1

We decided to use Scrum as our main development framework during the project. As we were a team of four we knew that we had the opportunity to achieve a decent implementation of this framework.

The 2020 version of the Scrum Guide states that a scrum team typically consists of 10 or fewer people [43]. Our team would be on the smaller side, which means communication inside the team would be crucial to stay productive. This was something we were ready for as the team members were well acquainted beforehand.

Another aspect which led us towards this process was our client. Signicat themselves use scrum and assured us that they would like to be involved and help us with the process.

3.9.2 Work Management Tools

The tools used to manage our work process was decided on by the client. We would use Confluence[9] to manage our documents and Jira[27] to keep track of our backlog, burndown-chart and sprints. For version control we would Signicat's internal GitLab.

3.9.3 Responsibilities within the team

E-mail correspondence: ***Thomas***
Responsible for communication between the team and outside elements.

Meeting documentation: ***Eric***
Responsible for the creation of meeting summons and writing meeting minutes.

Process documentation: ***Joakim***
Responsible for creating and organizing timesheets, status reports and other charts.

4 Results

This section will present the results of the project as a whole. It has been split to focus on the scientific findings, the current system in relation to the given requirements, and information about how the process was executed.

4.1 Scientific Results

4.1.1 Requirements for Research Question

To answer the research question we came up with the requirements presented in section 1. This section explains how we came up with these requirements.

4.1.1.1 Security

To keep data that we are handling secure, we have to keep several things in mind.

The first requirement states the user needs to be authenticated. This is often the first step in any security process and is how we identify the user. This is required so that we can check if they have the necessary authorization for what they are trying to do.

Secure handling requires that the data is not exposed at any point of the storage process. The fourth requirement states that the data must be kept secure when in transit and also once it is at rest in the database.

No matter how many safety measures are added to keep data secure, it is vital that we do not trust them blindly. The fifth requirement is that it must be possible to verify that any data stored has not been tampered with at any point.

4.1.1.2 Heterogeneous Data

The second requirement states that the heterogeneous data must be contained in some way. This is because we are dealing with diverse data, inputted by the user, where the form and shape of the data is largely unknown. In order to successfully handle this data across the different layers of the system architecture, each layer must find some way to encapsulate the data. This way we keep the user-defined data untouched, while dealing with container objects in the logic.

4.1.1.3 Multi-tenancy

To achieve multi-tenancy in way that is secure, it is imperative that we have authenticated the user, as stated in the first requirement. The third requirement is that we must still keep track of the owner of any data in the database. This is because there must be no possible way for a tenant in the database to be able to access others tenants data.

4.1.2 Benchmarking

At the beginning of development we wished to conduct some benchmarking on different HTTP frameworks for Java and various database solutions. We also wanted to explore how the different frameworks and databases integrated with each other. Another goal was to test which combinations were up to the task in regards to efficiency while having dynamic queries and good average response times. This was welcomed by the client despite the initial requirement that MySQL was to be used.

For frameworks we only looked at ones that have a large community and are widely documented. We decided to compare the established Spring Boot with the new challenger Quarkus, which is gaining in popularity.

For databases we knew that we wanted to explore both relational (SQL) and non-relational (NoSQL) databases to discover what would be best suited for the type of data we would be handling. The databases we looked into were:

- Non relational databases
  - MongoDB
  - Firestore (Firebase's database solution)
- Relational databases
  - MySQL
  - Postgres

4.1.2.1 Execution

We were able to connect Postgres, MySQL, MongoDB to Spring boot, and was able to setup MySQL through a Hibernate/persistence layer for Quarkus. The Firebase emulator proved challenging to work with within our environment due to our combined technology stack with Docker and Java. With time constraints, we decided to abandon testing of Firebase.

The benchmarking itself was conducted by setting up some basic endpoints using the HTTP methods GET, POST and DELETE. The GET methods included both getting a stored record by a indexed ID and getting a stored record by searching for a non-indexed value in it's heterogeneous data. We then conducted load testing on the different combinations of frameworks and databases.

4.1.2.2 Results

The full results from the benchmarking can be viewed in attachment 8.2.

4.2 Engineering results

At the end of development we have fully developed a REST API in Java and a frontend Demo Application in TypeScript. Both the API and the Demo Application are built using several of the design patterns mentioned in section 2.1

At the beginning of the project, Signicat (our client) provided several functional and non-functional requirements. These are detailed in the attached vision document (8.4). In this chapter we will go through these requirements and explain how we implemented them. Some requirements have changed during development as per the client's instructions.

4.2.1 API

The API has been developed as requested in Java 15 with preview enabled, which was the latest stable version at the start of development. The development has been done by following both the REST standard and Signicat's REST API guidelines.

> ”The team have done a nice job of creating a simple and clean API, that complies with Signicat REST guidelines”
> - Rune Synnevåg, Slack Message, 20.05.2021

4.2.1.1 Endpoints

The API contains all the requested endpoints related to the handling of Records. The initial requirements also mention specific Admin endpoints such as: ”Create customer”, but these were deprioritized during development by the client.

Create Records. The user is able to create a record and store it in their own collection in the database. This is accomplished by the customer supplying the data and metadata of the object they want to store in a JavaScript Object Notation (JSON) format. During creation the record is given an unique ID and generated system metadata is attached. The data of the customer will also be timestamped so that it can later be verified that the data has not been modified in any way.

Search Records. Users are able to search through their stored records by setting up queries containing search criteria. This can be used for simple search functionality or to filter through all their records. The search functionality supports both *”and”* and *”or”* logic meaning that several critera can be bound together and form complex queries. The search were specifically designed to only query the system- and customer metadata, but currently the core data can also be queried.

Get Records. Users are able to retrieve all information relating to a record by utilizing the record's unique id. When a record is retrieved in this way a check is also conducted to verify that the core

data of the record has not been modified. The result of this check is then displayed in the system metadata of the returned record.

Mark Records for deletion. Records stored in the database will automatically be given a Time to Live (TTL) of 3 years, after which the entry will be removed from the database. By using this endpoint the user can adjust the TTL to 30 days from when the endpoint is called.

4.2.1.2 Packaging

The requirement to build the API with Spring Boot and to release Docker images was achieved. Using GitLab's CI we automated generation and releasing of Docker images to GitLab's container registry.

Docker images are built when the master branch in the repository is updated, provided that all the other stages of the CI pipeline are passed. The template to build the Docker image is detailed inside a Dockerfile in the repository. The images are built during several steps in the CI pipeline.

The *"build"* stage builds an executable jar file of the API, then the *"test"* stage runs to see if there has been any breaking changes. If both the *"build"* stage and the *"test"* stage pass; the *"deploy"* stage will deploy the image by copying the jar from the *"build"* stage and push it to GitLab's container registry.

See attachment 8.1 for more details regarding packaing and CI.

4.2.1.3 Metadata

One of the requirements of the API was to include metadata defined by the user in addition to that generated by the system. The user is able to add in custom metadata to any records created, by simply attaching a JSON as the body of the request.

The initial requirement for the system metadata was to store:

- Time and date from Time-Stamping Authority (TSA)
- Delete grace period
- Mark for delete
- Valid until
- Validity of records related to timestamp
- Reference

Throughout the development process these requirements changed as we discovered MongoDB's TTL function served multiple purposes. The discovery of this functionality and resulting conversations between the development team and product manager lead to altering of the these requirements.

As TTL served multiple purposes, we removed the need for these criteria fields to be stored in system metadata:

- Delete grace period
- Mark for delete
- Valid until

4.2.1.4 Multi-module project

The API is split into several modules using Maven. We divided the project structure of the API into the following modules:

- core
- web
- mongoDB

This requirement was added during the development process when the client wanted the database integration of the application to be modular. They also wanted the project to have loose coupling and high cohesion. With the modules *"core"* and *"web"*, the API has the Model and Controller structure of a MVC, where as the Demo Application would function as the View in the context of the MVC design pattern noted in section 2.1.

Core
The *"core"* module have classes that serve as models. For instance, we have created the classes *"RecordRequest"* and *"RecordResponse"* which represent the data coming in to- and being returned from the API respectively.

The *"RecordRequest"* class would map the request body of a Hypertext Transfer Protocol (HTTP) request into a familiar format that the Digital Evidence Management (DEM) system could work with. While the *"RecordResponse"* class was made so that we could map the data generated by the system in addition to data retrieved from the database into a single response object to requests.

Web
The *"web"* module's function is to handle all logic in regards to the REST API. The *"web"* module

has controller classes like *"RecordController"* and *"AuthController"* to handle HTTP requests for the API's endpoints, as well as service classes like *"RecordService"* and *"TimeStampService"* to handle database operations and timestamping.

MongoDB
the *"mongoDB"* module consists only of two classes: *"RecordDAOImpl"* and *"SearchBuilder"*. *"RecordDaoImpl"* implements the methods specified in the Data Access Object (DAO) class in the Web module with it's own implementation that works with MongoDB databases. The DAO methods are uniformly written so that it is not dependent on a single database storage solution, and can easily use other database solutions by writing a implementation for the desired storage solution. The *"SearchBuilder"* class is a MongoDB specific helper class that helps with queries.

More on DAO can be found under section 2.1. The aforementioned modules are explained in greater detail in the System Documentation attachment.

4.2.1.5 Database Solution

The original requirement for the API was to use MySQL as the storage solution. However, after conducting the benchmarking (see section 4.1.2) it was agreed upon to change the database solution to use MongoDB as that proved to be more efficient and would facilitate the use of heterogeneous data.

4.2.1.6 OIDC

Spring Security is used to set up authentication with scope permission for most endpoints on the API. We also use Spring Security to filter the requests based on different authorization levels. The application or user requesting a resource from the API, will have to supply a valid access token along with any request for a resource.

All endpoints except "/auth", "/dem/actuatur/up" and "/dem/actuator/health" is protected with OAuth 2.0 and OIDC. The "/auth" endpoint is not protected as this is the proxy endpoint towards Signicat's authorization server to get a valid bearer token if provided client credentials are valid. The other endpoints mentioned that are not protected by OIDC are endpoints used for ascertaining if the API is up and running, and is most commonly used by Prometheus and other metrics scrapers.

4.2.1.7 Tenant Isolation

Isolation between the different users in the database is achieved by supplying each user a separate collection. When a user interacts with an endpoint they must send in a valid token in order to be given access, from this token the server is able to identify the user and thereby the collection to do

database operations towards. Now the server targets the specific collection belonging to the user. It is also safe to remove tenants from live systems as this would not affect other tenants.

4.2.1.8 Encryption at rest

The database currently set up with the API is using MongoDB's Community Edition, which does not have encryption at rest. By making the switch to MongoDB's Enterprise edition, the data would be encrypted.

4.2.1.9 Test Coverage

The requirement for testing coverage was to have 90% coverage. We currently have achieved testing coverage of 85%. This is accomplished by both unit tests and integration tests.

4.2.1.10 Development Tools

The API follows the client's coding standard which is enforced by using the Checkstyle development tool.

4.2.1.11 Documentation

The API has documentation supplying information about how each endpoint functions and examples of usages. The API has both Swagger and ReDoc documentation which are generated on build through the use of an OpenAPI Specification (OAS) file that is produced by the plugin SpringDoc.

4.2.1.12 Metrics and Logging

The API uses the Spring Actuator framework to expose endpoints with default metrics which can be scraped with tools like Prometheus. We have set the API to use Log4j to print logs instead of "System.out.println()", but have not connected Log4j to any centralized logging solution at this time as it was not set as a priority by the product manager.

4.2.2 Demo Application

4.2.2.1 Programming language

The requirement to use JavaScript/Hypertext Markup Language (HTML)/Cascading Style Sheet (CSS) is partly altered. The project is using HTML and CSS, but replaced JavaScript with TypeScript instead.

This was done to gain the advantages and features of a compiled language and had no seemingly negative consequences. TypeScript is just a superset of JavaScript which compiles down to browser friendly JavaScript. By using TypeScript instead, we gained several advantages as described in section 3.7.1.

As stated in the Requirement Document (see 8.3), we were also to choose a HTML framework and a CSS framework.

We choose to use React as the HTML framework, since every member on the team had prior experience with it, and to reduce time otherwise spent on learning another framework.

Since Signicat had their own style guides, but no framework. We saw that if we were to use a CSS framework like Bootstrap, Materialize or Material Design, we would need to override most of the styling from those frameworks to fit with Signicat's style profile. Because of this we wrote the CSS classes from scratch instead.

4.2.2.2 Packaging

We were able to fulfill the requirement to release containers. On GitLab CI's final stages on updates to master, the GitLab CI runner will produce a production build of the Demo Application and build a Docker image which is pushed to GitLab's container registry.

4.2.2.3 Unit test code coverage

For the Demo Application we were successful in reaching the required minimum of 85% coverage. Our application had a coverage of 95.87% for statements, 85.59% for branches and 89,6% for functions. With an average coverage of 90,35%.

See attachment 8.1 for generated coverage report.

4.2.2.4 Linting

The Demo Application is set up to use Eslint for all linting purposes, and uses Prettier for style formatting. Eslint is set up with a strict ruleset, where errors will not compile. It will also output to console if there are any warnings. More details about Eslint and Prettier can be found in the System Documentation attachment(see 8.1), and in section 3.7.3.

4.2.2.5 Search functionality

The requirement for the Demo Application, was for it to be able to showcase the powerful search functionality that the API had. We were successful in implementing most of the functionality that the *"/query"* endpoint that the API had to offer. But did not implement the *"OR"* conditions into the Demo Application as we felt it was not intuitive enough for the user of the GUI.

We also implemented the search to work together with page-able requests, so that the Demo application could have *"lazy loading"* functionality.

See attachment 8.1 for a greater explanation of the features and functions of the search endpoint.

4.2.2.6 Test Data Generation

During the development of the Demo-application, a new requirement was introduced to have the user be able to generate randomised test data, so that the user could test the system with more data. This goal was achieved, but could also be a part of further development to have the user specify more settings in regards to the generating of data.

4.2.2.7 Authentication

The requirement for needing Authentication for the Demo Application came very late in the project. The team together with the product manager saw the need to have some sort of authentication since the Demo-application would be deployed to Amazon Web Services (AWS).

The Demo-system partly achieved this goal. This is due to the OAuth 2.0 client-credentials authentication flow (see 2.3.2) used in the API was not intended to be used in this manner. Through discussions with Tech Lead and Product Manager, we came to a conclusion that since the system in AWS only was a test environment, it would be acceptable to have some security flaws. The authentication requirement for the Demo-appliction was more of an effort of limiting access, rather than securing it entirely.

Furthermore, since the API was the main product, it did not seem right to have the API make

compromises when the Demo-application was only intended for a test environment, and not a production environment. More about the details regarding authentication and security can be found in the system documentation.

4.2.2.8 WCAG

The requirement in the vision document was full support for WCAG 2.1. This requirement was mostly fulfilled by creating a small React component Library that fulfilled WCAG 2.1, which we then reused through the entire demo application. Fulfilling these requirements also meant that we had to do some adjustments to Signicat's already established style profile. By using the JSX A11y plugin for ESlint, we made the code meet many of the requirements immediately on compile. As compile in React was set up to be on save, we got immediate feedback on the code. This includes all the html-attributes that WCAG 2.1 requires.

The demo application mostly fulfilled it's goal of full WCAG 2.1 compliance. The most important items like offering support for screen readers and more was implemented through correct use of html-attributes and tab navigation.

4.3 Deployment

The initial requirement was for the deployment to target Kubernetes, but during the development Signicat's infrastructure team decided that this version of the system should be deployed with Nomad.

When the master branches are updated, GitLab Continuous Integration (CI) runs a pipeline that in its final stage deploys a Docker image of the build to the GitLab Container Registry. The API and Demo Application both contain .hcl files which are used to get both system running and interacting in Nomad. These .hcl files are used by Nomad to create containers from the images stored in the container registry's.

For the actual deployment these files are run in a different repository belonging to Signicat where Terraform is used to deploy the application in Amazon Web Services (AWS).

The current deployment is only available inside Signicat's internal network.

4.4 Encryption in transit

Encryption in transit is partly achieved, the API and Demo-application we deployed to run on Amazon Web Services (AWS) uses HTTPS, but due to a certificate name error, the certificate is rendered invalid. The certificate is set up and supplied by Signicat, so we have notified Signicat to correct this. So when the certificate is valid, both services will have encryption in transit.

4.5 GDPR Compliance

The system as of now does not store or log any personal information as the actual user accounts are managed by Signicat. As the project did not reach production during development no other consideration regarding the General Data Protection Regulation (GDPR) has been considered.

4.6 Merge to master

The initial requirement to have merge requests merged by employees was quickly changed as the pace needed for creating a foundation for the API and the Demo Application required more haste. We changed this requirement to have another member of the team approve merge requests.

4.7 Snyk

Both the API and Demo-application is integrated with Snyk.io. However, they both have some vulnerabilities reported by Snyk. At the time of writing this report, there is no patch available yet to remedy these vulnerabilities.

4.8 Missing requirements

- The feature of records being able to have relations to each other has not been implemented in our solution.
- The integration with Sonarcloud has not been implemented.
- The integration with Signicat Billing was removed by the client.
- Versioning has not been conducted to the degree specified.
- Centralized logging has not been conducted to the degree specified.

4.9 Administrative Results

4.9.1

As stated in section 3.9.1, we chose to use Scrum as our development framework.

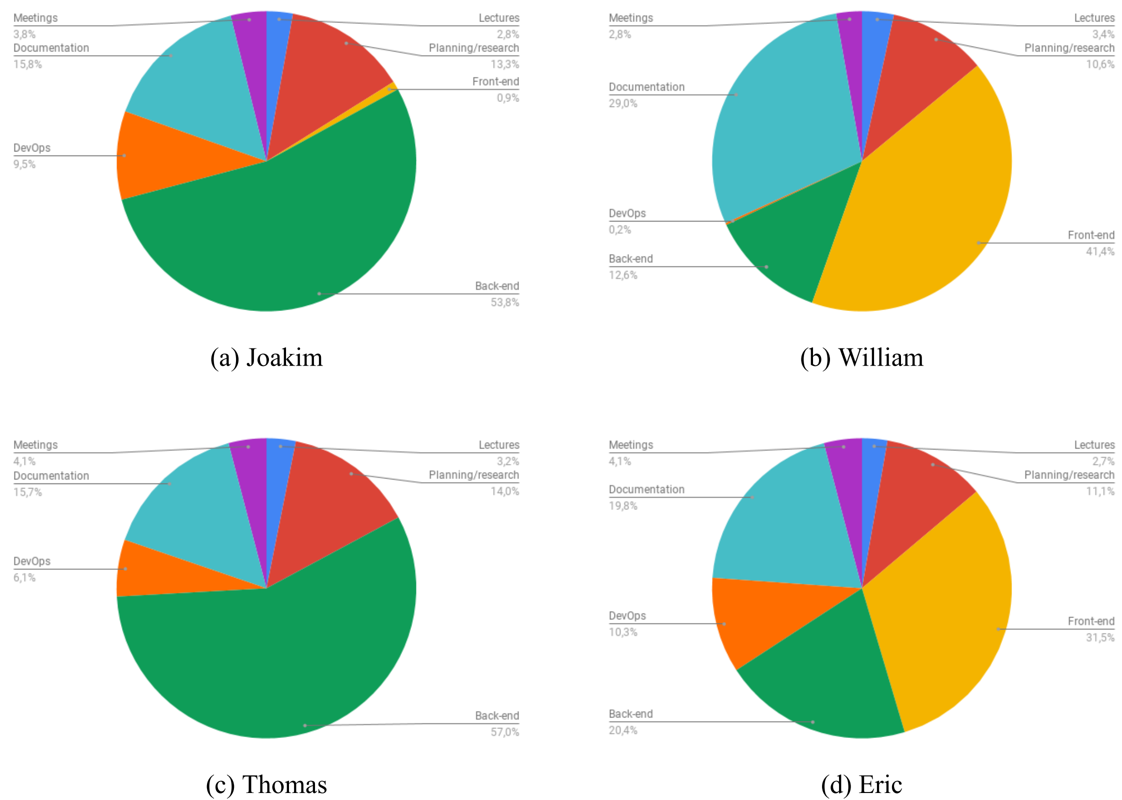
The biggest deviation from the standard methodology was the lack of a dedicated scrum master. The closes thing we had to this role was Tor Even Dahl, who was our primary contact with Signicat. He would weigh in on what should be prioritized next, arrange meetings and join our sprint reviews. However much of the actual scrum process was largely up to us to maintain, with members from Signicat having access to all our planning.

We started the project by setting up a backlog, after which we received feedback and approval from Signicat. During the development time we conducted in total five sprints with each one lasting between two to three weeks.

We would begin each sprint by planning out which features and qualities we wanted to accomplish in the time period. After setting up all the issues for the sprint we conducted a ”planning poker” session, where each member wrote down how long they thought it would take to accomplish each issue. After everyone revealed their estimates we all agreed on a number of hours to assign to the given issue.

After each sprint we would conduct a sprint review with members of Signicat staff and present our results. We would then have a discussion on possible changes and future priorities. After the review we would go together as a team and reflect on the sprint. We based these discussion around the retrospective technique of the 4L’s (Liked, learned, lacked, longed for)[22], having each member write down some aspects for each category. We eventually added ”Disliked” as a fifth column.After everyone had written their thoughts, we would read through each category and discuss what general themes we were observing. At the end of each retrospective we would write down a list of what we should improve upon for next time.

4.9.2 Work distribution



(a) Joakim (b) William

(c) Thomas (d) Eric

Figure 9: Time distribution per team member

We found that the team originally split into two teams, with one focusing on the API, while the other focused on the Demo Application. As time progressed it made sense for all members to work more together on the application as a whole.

More information about the process and each members timesheet can be found in attachment 8.5, Process Document.

5 Discussion

In this section we will reflect on the results achieved and provide explanations on what went well and what went less well. This section is split in the same categories as those presented in Section 4.

5.1 Scientific Discussion

5.1.1 Requirements for Research Question

We defined the requirements for the research question primarily based on the various conversations and technologies we encountered during the course of the project. We believe that the requirements are accurate to the challenges of the task, however, we could have done more research to strengthen our case or find weaknesses in our logic.

5.1.2 Benchmarking

At the beginning of development we wished to conduct some benchmarking on different Java framework- and database solutions. We also wanted to explore how the different frameworks and databases integrated which each other. Another goal was to test which were up to the task in regards to efficiency while having dynamic queries and good average response times. This was welcomed by the client despite the initial requirement that MySQL was to be used.

Our initial goal was to connect all four databases to both Spring Boot and Quarkus. We quickly saw that we had underestimated the amount of time it would take to implement all the databases with both frameworks, and after some discussion with our supervisor, the product owner, and the tech lead, we decided that conclusive benchmarking was too big a task for the time we had allotted. We then decided to make our report and final decision at the end of sprint, even if we were not entirely satisfied with the data we were able to ascertain in this time.

The main area of concern was how poorly Quarkus performed in most of our tests compared to Spring Boot. The discrepancy between the two led us to suspect that our implementations may not have been sufficiently comparable. Nonetheless, this benchmarking phase helped us identify which combination of framework and database felt more natural for the way we envisioned the system, which is why we ultimately recommended using Spring Boot and MongoDB.

5.2 Engineering Discussion

The original hope was to complete the product in its entirety, but we do not believe this was really achievable with the time allotted for this assignment. In our opinion, changing the scope to more of a test environment made sense. It also means that potential customers can experiment with our system and help shape the final product by providing feedback.

Even though we did not meet all requirements specified in the vision document, we are satisfied with meeting the core requirements of DEM: TSA, OIDC and Multi-tenant isolation. These core requirements serves as the foundation of DEM, and will function as building block for any further development. These concepts was hard to both comprehend and integrate with the API.

In this section we will discuss in more depth different aspects of the system.

5.2.1 Development Tools

We are satisfied with the development environment we have set up for the API and the Demo application. By using the strict Checkstyle and Eslint plugins, we were able to reduce the number of bugs produced and increase the readability of the code. We also used Docker-compose to set up MongoDB and Prometheus with the same configuration for all members of the team, which saved us a lot of time since we didn't have to configure it on each member's machine. GitLab CI was helpful to avoid including buggy code in the master branch.

The different tools and technologies we employed during development did turn out be very beneficial over time. For the API, we had plugins to run both integration tests and unit tests when executing build commands, which led to errors early in development. Using Maven as the project's package manager also made it easy to manage dependencies within the different repositories. For the Demo Application, we integrated Prettier in addition to Eslint to handle style formatting and linting, which increased readability and unified the code.

5.2.2 Query

We are happy that we were able to construct a powerful query endpoint on the API, which let the user have a fine grain query by sending in lists with ”AND” and ”OR” conditions that should be met. Within the aforementioned lists they could match criteria with operators such as:

- Equals (eq)
- Greater than (gt)
- Greater than or equals (gte)

- Less than (lt)
- Less than or equals (lte)
- In (in)
- Not in (nin)
- Regex (regex)

5.2.3 Deployment

We were able to meet the requirements in terms of releasing docker containers and deploying to Amazon Web Services (AWS). The process of releasing docker containers through GitLab and deploying to AWS was time consuming, as we did not have much knowledge and experience prior to this.

We were able to meet the requirements of creating docker containers and deploying these in AWS. The process of releasing Docker containers via GitLab CI and deploying to AWS was time consuming as we did not have much prior knowledge and experience.

Deploying to AWS was unfortunately more complicated than first imagined. Signicat has not whitelisted GitLab CI to deploy to AWS, which was our original intention. In order to allow this, Signicat first had to do an evaluation to determine if they wanted to allow it at all.

The suggested solution was to use the alternative CI solution in Jenkins, which was allowed to deploy to AWS.

We did not elect to rewrite our pipeline in Jenkins because we had already invested a lot of time in our GitLab integration. Also, we were much more familiar and comfortable with GitLab than with Jenkins. It was our general impression that GitLab would be approved and whitelisted, but we could not rely on this to be prioritized in time for our needs. We instead elected to use a workaround so that we could still use the solutions we had already created.

This workaround involved integrating our Nomad job files with another of Signicat's repositories, which already setup to deploy to AWS. The problem with this solution, was that we had to manually trigger a CI pipeline for another repository to have AWS pull the newest containers. So in effect, we were only able to achieve Continuous Integration and not complete Continuous Deployment in our solution.

There were several routes we could go for deploying our systems to AWS, and we believe our current solution is not ideal or perfect. There were several ways to tackle the CD problem, but due to some pressure on having a running instance for customers of Signicat deployed to AWS within the end of the project, we believe we made the best of the situation.

Our ideal solution was the one that we wrote at first, to have GitLab handle both CI and CD. With our initial solution there would be no coupling/dependency to another project, and our own repositories would automatically handle both deployment and have AWS restart and pull the latest images. All could be setup and maintained from a single stage with GitLab CI.

GitLab
We had to finder Docker base images that would fit each stage of the pipeline. To test if a docker image had the correct tools and software needed we had to run it through GitLab and wait for the job to pass. Some stages required several tools, for instance the stage "build-pages" as noted in the System Documentation(see 8.1), required having Maven 3.8, Java 15 with preview and a small Linux distribution like Buster to run shell commands. Although we were able to make stages that would fulfill our CI / CD needs and meets the requirement, we note that the stages are poorly optimized. We could have done more research into finding docker images with a smaller footprint which in turn would decrease the time needed for each stage to complete.

Versioning
Our GitLab containers are tagged with the tag "latest", and should have been more specific in order to easier rollback solutions if there should arise problems with a new deployment.

5.2.4 Encryption in transit

The requirement of encryption in transit is something we did not have to put any effort in acquiring for our deployed instances on AWS. The environment on AWS was already set up to use HTTPS. It should be noted that there is a problem with the certificate name already set up for the AWS services, which in effect makes this requirement only partially fulfilled. We have notified Signicat to correct this certificate error, and are confident that encryption in transit will be acquired when the error is fixed.

5.2.5 Encryption at rest

We found out that to have the database encrypted, we only had to make the switch from MongoDB's Community edition to MongoDB's Enterprise edition. This was put on hold from Signicat, as the billing would start from the second we made the switch to the Entreprise edition.

Seeing as there were no need for an encrypted database in the developing stages, it made sense to us to not push any further on the issue. We consider this requirement as essential when the product reaches a production state, but for the demo environment, we felt that it was not necessary yet.

5.2.6 OIDC Flow

In the start of the project, there were no initial thoughts that the Demo Application would need authentication. However, as the project details become more clear as we developed the DEM system, we saw a need for securing the Demo-application as well. At that point, the client-credential flow (see 2.3.2) was well integrated with the API, and we later realised that this flow would not work well for both Single Page Application (SPA) and machine to machine communication like the API needed.

We set up our API to use OAuth 2.0 client credentials protocol flow as this was the best fit for machine to machine interaction[30]. This worked great for the API as a stand-alone product. The problems with client-credential flow are that no refresh tokens are granted and the protocol is not intended to be used in a user event-driven manner, but rather for machine-to-machine interaction.

To safely secure any frontend solution like our Demo Application with persistent log-in with this protocol would be troublesome, if not next to impossible. [30] This is because the client-credentials consists of a client id and a client secret known as authorization grant, and by providing the authorization grant to an authorization server would return a access token.

We initially stored the credentials in memory only for the Demo Application, but later came to the decision to store the credentials in ”localStorage” to favor a good user experience, in order to not have the user log in on page refreshes. By the storing either the authorization grant or the access token in ”localStorage”, the application becomes vulnerable to Cross Site Scripting (XSS) attacks as the application would supply an attack vector. We explained this security flaw in greater detail in the System Documentation (see 8.1).

Our research showed that the OAuth 2.0 PKCE flow (see 2.3.3) was the best fit for SPA's, where the protocol would provide us with refresh tokens to use within the Demo Application [32]. However, it was always the thought that the API was the main product that would reach production, while the Demo Application would only be used as a demo environment. Because of that fact, we did not want the API make any compromises on behalf of the Demo Application. Through discussions with Signicat, we came to the conclusion that it would be acceptable for the demo-environment to have some security flaws for the time being. For further work we would look into more of a Active-Directory solution for safely securing the Demo Application.

It is not ideal to have a insecure system even though it is only intended for a demo environment. This is something that we would wish to correct if we had more time, and is definitely something that we would put in the further development column to correct.

5.2.7 Other requirements not met

Some requirements was not implemented at all as it was not set as a priority by the product manager, and there was simply just not enough time. We do not see the requirements that we were not able to meet as crucial to the MVP, but rather that they are essential for DEM to reach a production state. We also found that the missing requirements was something that could be added later on and would not require any refactoring of the project structure or setup.

The requirements we were not able to meet:

- Sonarcloud integration for both API and Demo-application
- Integrate with Signicat billing solution
- Centralized logging for the API
- Versioning.

5.2.7.1 Snyk vulnerabilities

We have integrated Snyk into both the API and the Demo-application, but there are some vulnerabilities that has been reported by Snyk that we were not able to fix. Some of the dependencies library used that were vulnerable i.e Spring Boot, React scripts were so integrated into the system that there were no simple operation of just replacing the dependency with another. We opted for the option to put the vulnerabilities in a Snyk ignore file for thirty days, and have Snyk let us know about the vulnerabilities after the snyk ignore expiration date. This option is something we deem as a valid solution, as the developers of the libraries and frameworks will often need time to write patches to fix those vulnerabilities. Snyk will notify us if there are any patches available on the expiration date of Snyk ignore issues.

5.2.7.2 Reference between records

We did not have time to implement the functionality of having reference between records, as this is a design discussion as well as a technical implementation. We saw that implementing a reference between record was not as simple as just setting a reference between one record to another. The design discussion is a matter of if we should have full transitive relation between records, just a one way binding, or a two way binding between records. This topic of relationship required further research. We want to note that this is a requirement that can be added later on without requiring any, or little refactoring, as we had it in mind when we designed the database model and set up the API.

5.2.7.3 Using POJOs and Records

Something we wanted to use in this project was the Java feature of Records, not to be confused by our ”Record” entity. Records are a new type introduced in Java 14 as a preview, with the intention to avoid much of the boilerplate code that is required in normal POJOs[21]. This suited our data very well and we were able to use it efficiently through most of the project. When we wanted to implement HATEOAS to our API we found that this was not easily integrated without using POJOs. Spring has a built in solution for HATEOAS, but this does not have support for Records at this moment. To solve this we had to redesign the system to translate between Java records and POJOs when relevant. Ideally we would have liked to be consistent with the types throughout.

5.3 Administrative Discussion

5.3.1

At the end of the day we feel that the use of Scrum worked well for this project. This is especially because we had a client which wanted to stay involved in the process throughout and who had experience in working this way. By splitting the development into several sprints we were able to break down the assignment to smaller parts and tackle them one at a time.

We did feel very self-sufficient as a team, especially so because of the lack of a fully dedicated Scrum master. This absence is definitely a major one for our methodology to follow the Scrum framework recommendations.

We did find that the time spent to execute the spring ceremonies often exceeded what we initially planned. This was something that improved over time, but we could still have been more efficient in how we went about the planning and retrospectives of the sprints.

5.3.2 Teamwork

As with all large systems there are the three main challenges [29]:

1. Complexity
2. Lack of insight and overview
3. Communication challenges

Throughout developing DEM we faced all of these challenges as the size of the system grew. Since we developed both a API with multiple modules and a demo application, our system became large and complex (1), and it was hard for everyone to have insight and overview of the system(2). We had a Mini workshop in the middle of our Bachelor project as an effort to get more overview and insight into the parts of the system that people had not worked on.

We had some difficulties regarding to communication(3) while developing our system and writing our bachelor thesis. This was mostly due to our team were located in different places. Three of the team members were located in Trondheim, while the last team member were located in Oslo. The difficulties of working from different places was that we lost track of what others was working on, and how the progression was for certain tasks. The threshold was higher for questions and smaller conversations when we were not in the same place working together.

Furthermore, because Covid-19 infection rates was on the rise in the midst of our Bachelor project, we were prohibited from gathering at both NTNU and at Signicat's offices for a month. During

this time, we tried relying as best as we could on tools and technology like Discord for voice and video conversations, daily stand ups to gather a status on what people were working on and if they had any problems.

We noted these communications issues on our Sprint retrospectives, and worked on being more available on voice chats, asking each other about issue status, and also worked on writing more detailed issues with sub-goals so that it would be easier to know what to do, and to follow the progress of issues more easier. We also became more strict with using Jira early on as we saw the need for such a tool for asynchronous communication. Moreover, to alleviate the communication problem, we also made use of sessions with pair programming from time to time. This was particularly helpful when it came to issues that were difficult to comprehend or implement.

5.4 Societal Perspective

The DEM system is made so that it is possible to have both single-tenant and multi-tenants. Single-tenancy is theoretically more secure because you do not have multiple users within the same database, and can therefore guarantee user isolation. In a multi-tenant solution the users are all inhabiting the same database. Users are isolated from each other in a multi-tenancy solution, but there is always the risk of containment breaches in the isolation.

Ideally we would always use single-tenancy, but the cost of running multiple instances of database makes this economically unsustainable. In a realistic situation one has have to weigh the degree of security up against the cost of running the system.

With the emergence of GDPR and all of its requirements and rules, a need for storing sensitive data in a secure manner emerges. DEM tries to cover that need by supplying a service which can store the data securely, verify that data has not be breached, and has not been tampered with. DEM aims to be a general evidence management solution that would fit a wide variety of industries which have the functional role as data controllers. Both customers of Signicat, and Signicat themselves has expressed the need for such a solution, and will start to use it when it is production ready.

5.5 Professional Ethics

As the Digital Evidence Management (DEM) system's main purpose is to store sensitive information, there are several requirements that need to be met as a data processor before the system can be used in production.

We asked the product manger Tor Even Dahl for DEM to explain the role of data processor and the responsibility that comes with it:

> ”The most important keyword for a data processor is trust. Customers need trust that the data processor processes the the customers data in regards to laws and regulations and does so in a secure manner. The data processor would need to have good processes internally and security in the way it handles the data without intrusion, deleting data in accordance with defined rules and etc.”
> - Tor Even Dahl, Slack message, 18.05.2021

From a small extract of the data controller agreement[11], we can find many points that need to be considered when developing Digital Evidence Management:

- The data processor acts according to instructions
- Confidentiality

- Security of processing
- Use of sub-processors
- Transfer of data to third countries or international organisations.
- Notification of personal data breach
- Erasure and return of data
- Audit and inspection

The data processor contract is very detailed, but for edge cases which the contract does not cover, the team of developers, as well as Signicat as a company, should always focus on having good professional ethics, following laws, and when in doubt make use of organisations like Datatilsynet, NITO, IEEE and ACM.

As we mentioned in previous chapter, the demo environment has some security flaws regarding to storing credentials in ”localStorage” on the Demo-application. If the Demo-application should ever be used in a production environment, it is crucial that these security flaws be corrected. It is part of the responsibility that comes with the role ”Data Processor” and is also not seen as good practice within the profession to knowingly deploy a security flawed system to production.

In addition to security, it is also important that the application is made to be accessible. We made an effort to fulfill the WCAG 2.1 requirement with necessary alt texts, tab indexes, key listeners, labels, correct color ratio etc, so that the system should be accessible to all users with or without disabilities.

As we can see from extract, the core requirements: ”Multi-tenant user isolation”, ”Encryption in transit”, ”Encryption at rest”, ”Verification of data tampering” is quite relevant when it comes to fulfilling the role as a data processor.

6 Conclusion

Securely handling heterogeneous data in a multi-tenant solution is possible if one considers all the requirements that such a solution demands. We identified 5 requirements to satisfy our research question:

1. Authenticate the user.
2. Contain the heterogeneous data.
3. Manage ownership of data in the database.
4. Keep the data secure in transit and at rest.
5. Verify that the data has not been modified.

If any of these requirements are not met, one could not claim to have developed a sufficient solution. In this report, we have explored these requirements and how we have chosen to address them in DEM.

We believe that our solution fulfills our research question. This is based on the requirements we laid out from the research question, the requirements Signicat gave us with our assignment, and the feedback we have gotten from both the employees of Signicat and potential customers. DEM is not ready for production yet, but it is at a stage where most of it's requirements are met and it is deployed to a test environment.

It is at a point now where it is desirable to involve end users and gather feedback for possible improvements and new features. DEM has already been shown to some potential customers and has been enthusiastically received. The next stage for this product will be allowing these customers access to the Demo Application so they can freely test its capabilities.

6.1 Further work

For any further development of the application we recommend that the following points are added or considered:

- Involve end-user
- User testing
- Add feature to provide relations between different records.
- Change the database to use MongoDB Enterprise edition so that the database will be encrypted.
- Redesign the Demo Application login to correctly use an authorization code flow.
- Add proper refresh tokens to the Demo Application.
- Connect the system to Signicat's billing systems.
- Connect the system to Signicat's centralized logging system.
- Set up detailed metrics.
- Integrate SonarCloud for automatic code reviews.
- Improve test coverage on the API (currently the API is at 85% coverage, the requirement is 90%).
- Upgrade the API to Java 16.
- Better handling of date objects (the current method of parsing date objects work, but are not ideal).

7 References

[1] *ACID properties of transactions*. URL: https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions (visited on 05/19/2021).

[2] *ACID Transactions in MongoDB*. URL: https://www.mongodb.com/basics/transactions (visited on 05/19/2021).

[3] Alexandra Altvater. *Gradle vs. Maven: Performance, Compatibility, Builds, More*. URL: https://stackify.com/gradle-vs-maven/. retrieved 08.03.21.

[4] *Article 21*. URL: https://www.eba.europa.eu/regulation-and-policy/single-rulebook/interactive-single-rulebook/5504 (visited on 05/19/2021).

[5] *Article 5*. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2018.069.01.0023.01.ENG&toc=OJ:L:2018:069:TOC (visited on 05/19/2021).

[6] *Authorization Code Flow with Proof Key for Code Exchange(PCKE)*. URL: https://auth0.com/docs/flows/authorization-code-flow-with-proof-key-for-code-exchange-pkce (visited on 05/18/2021).

[7] Chris Brook. *SaaS: Single Tenant vs Multi-Tenant - What's the Difference?* URL: https://digitalguardian.com/blog/saas-single-tenant-vs-multi-tenant-whats-difference (visited on 05/20/2021).

[8] *Client credential flow*. URL: https://auth0.com/docs/flows/client-credentials-flow (visited on 05/18/2021).

[9] *Confluence*. URL: https://www.atlassian.com/software/confluence (visited on 05/17/2021).

[10] *Data Access Object*. URL: https://en.wikipedia.org/wiki/Data_access_object (visited on 05/19/2021).

[11] Datatilsynet. *Databehandleravtalen på engelsk*. URL: https://www.datatilsynet.no/rettigheter-og-plikter/virksomhetenes-plikter/databehandleravtale/hvordan-lage-en-databehandleravtale/hva-ma-en-databehandleravtale-inneholde/ (visited on 05/18/2021).

[12] *Docker overview*. URL: https://docs.docker.com/get-started/overview/ (visited on 05/20/2021).

[13] Docker.com. *Docker compose*. URL: https://docs.docker.com/compose/. retrieved 08.03.21.

[14] *Encryption at Rest*. URL: https://docs.mongodb.com/manual/core/security-encryption-at-rest/ (visited on 05/20/2021).

[15] Eslint.org. *About Eslint*. URL: https://eslint.org/docs/about/. retrieved 08.03.21.

[16] Facebook. *React.js*. URL: https://reactjs.org/. retrieved 08.03.21.

[17] Roy Fielding. *Representational State Transfer*. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. retrieved 07.04.21.

[18] Figma. *Figma*. URL: https://www.figma.com/. retrieved 08.03.21.

[19] Apache Software Foundation. *What is Maven?* URL: https://maven.apache.org/what-is-maven.html. retrieved 08.03.21.

[20] Günther Franke. *http://w3sdesign.com/?gr=s05ugr=struct*. (Visited on 05/19/2021).

[21] Brian Goetz. *JEP 359: Records (Preview)*. URL: https://openjdk.java.net/jeps/384.

[22] Mary Gorman and Ellen Gottesdiener. *The 4L's: A Retrospective Technique*. URL: https://www.ebgconsulting.com/blog/the-4ls-a-retrospective-technique/ (visited on 05/17/2021).

[23] Network working group. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*. URL: https://tools.ietf.org/html/rfc3161. retrieved 06.04.21.

[24] Anna Hannover. *Introducing Qualified Time Stamp Authority service from Signicat*. URL: https://www.signicat.com/resources/introducing-qualified-time-stamp-authority-service (visited on 05/19/2021).

[25] *How does TLS work?* URL: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/ (visited on 05/18/2021).

[26] MongoDB Inc. *The database for modern applications*. URL: https://www.mongodb.com/. retrieved 08.03.21.

[27] *Jira Software*. URL: https://www.atlassian.com/software/jira (visited on 05/17/2021).

[28] Takahiko Kawasaki. *Understanding ID token*. URL: https://darutk.medium.com/understanding-id-token-5f83f50fa02e (visited on 05/19/2021).

[29] Grethe Sandstrak Kirsti Berntsen Geir Ove Rosvold and Marthe Liss Holum. *FS 01 Intro Systemtenking*. https://ntnu.blackboard.com/webapps/blackboard/execute/content/file?cmd=view&content_id=_1205521_1&course_id=_22717_1. Lecture slide from TDAT3002. Retrieved 18.05.21.

[30] Microsoft. *The OAuth 2.0 Authorization Framework, RFC*. URL: https://tools.ietf.org/html/rfc6749. retrieved 06.04.21.

[31] *MVC: Model, View, Controller*. URL: https://www.codecademy.com/articles/mvc (visited on 05/19/2021).

[32] *OAuth PCKE flow*. URL: https://datatracker.ietf.org/doc/html/rfc7636 (visited on 05/17/2021).

[33] OpenID. *OpenID Connect*. URL: https://openid.net/connect/. retrieved 06.04.21.

[34] Tamara Pattinson. *Relational vs. non-relational databases*. URL: https://www.pluralsight.com/blog/software-development/relational-vs-non-relational-databases (visited on 05/19/2021).

[35] Prettier.io. *Prettier*. URL: https://prettier.io/. retrieved 08.03.21.

[36] *Revised rules for payment services in the EU*. URL: https://eur-lex.europa.eu/legal-content/EN/LSU/?uri=CELEX:32015L2366 (visited on 05/19/2021).

[37] Scrum.org. *Scrum Product Backlog*. URL: https://www.visual-paradigm.com/scrum/what-is-product-backlog-in-scrum/. retrieved 08.03.21.

[38] *Separation of concerns*. URL: https://en.wikipedia.org/wiki/Separation_of_concerns (visited on 05/19/2021).

[39] *Signicat Express APIs*. URL: https://www.signicat.com/express-apis (visited on 05/16/2021).

[40] *Single-Tenant LMS vs Multi-Tenant LMS: A Question of Security*. URL: https://www.peoplefluent.com/blog/learning/single-tenant-lms-vs-multi-tenant-lms-security/ (visited on 05/19/2021).

[41] *Software design patterns*. URL: https://en.wikipedia.org/wiki/Software_design_pattern (visited on 05/19/2021).

[42] Spring.io. *Spring Security*. URL: https://spring.io/projects/spring-security. retrieved 06.04.21.

[43] Ken Schwaber Jeff Sutherland. *The 2020 Scrum Guide*. URL: https://scrumguides.org/scrum-guide.html. retrieved 17.05.21.

[44] Lidong Wang. "Heterogeneous Data and Big Data Analytics." In: *Automatic Control and Information Sciences* 3.1 (2017), p. 1. ISSN: 2375-1630. DOI: 10.12691/acis-3-1-3. URL: http://pubs.sciepub.com/acis/3/1/3.

[45] *What a Relational Database Is*. URL: https://www.oracle.com/database/what-is-a-relational-database/ (visited on 05/19/2021).

[46] *What is a Non-Relational Database?* URL: https://www.mongodb.com/non-relational-database (visited on 05/19/2021).

[47] *What is HTTPS?* URL: https://www.cloudflare.com/learning/ssl/what-is-https/ (visited on 05/18/2021).

7.1 Personal Communication

- Tor Even Dahl, Product Manager, Team Signature at Signicat.
- Steinar Knutsen, Tech Lead, Team Signature at Signicat.
- Rune Synnevåg, VP Lead Architect, Signicat

8 Attachments

8.1 System Documentation

System documentation: Digital Evidence Management

Joakim Moe Adolfsen William Jarbeaux Thomas Bakken Moe
Eric Younger

Spring 2021

TDAT 3001 - Group 109
version 1.0

Audit history

Date	Version	Description	Author
13.01.2021	0.1	Initial structure setup	Thomas Bakken Moe
12.05.2021	1.0	Filled out details. Added project structure, diagrams and installation	Joakim Moe Adolfsen, Thomas Bakken Moe, Eric Younger

Contents

1 Introduction **9**

2 Architecture **10**

3 Project structure **11**

3.1 Backend 11

3.1.1 Root structure 11

3.1.2 Core 13

3.1.3 MongoDB 20

3.1.4 Web 21

3.2 Frontend 24

3.2.1 Root structure 24

3.2.2 Src 26

4 Diagrams **32**

4.1 API 32

4.2 Demo Application 35

5 Database model **36**

6 Server services **37**

6.1 Adding a record 37

6.2 Retrieving a record 37

6.3 Marking a record for deletion 38

6.4 Retrieving specific records 38

6.5 Others 39

7 Security 40

8 Installation and running 44

8.1 API 44

8.1.1 Synopsis 44

8.1.2 Pre-requisites 44

8.1.3 Error handling for database 44

8.1.4 Build project 45

8.1.5 Run project 45

8.1.6 API-Documentation 45

8.1.7 Metrics 46

8.1.8 Profiles 46

8.1.9 Switching databases 47

8.1.10 Static code analysis tool: Checkstyle 47

8.1.11 Dockerfile 47

8.2 Demo-application 48

8.2.1 Synopsis 48

8.2.2 Pre-requisites 48

8.2.3 Run application 48

8.2.4 Run tests 49

8.2.5 Build production app 49

8.2.6 Environments 49

8.2.7 Git-Hooks 50

8.2.8 Linter and style formatting 50

8.2.9 Dockerfile 51

8 ATTACHMENTS - System Documentation 68

9 Continuous integration and testing 52

9.1 Backend 52

9.2 Frontend 54

9.3 Testing 56

9.3.1 API - Unit and Integration Tests 56

9.3.2 Demo Application - Component Tests 56

9.3.3 Snyk 57

List of Figures

1 High level figure of the architecture. 10

2 The root structure for the DEM API. 11

3 The contents of the core module 13

4 The contents of the record folder 13

5 Record, the data structure that gets stored in the DB. 14

6 Record request, this is the data the customer provides when requesting to post a new record to DEM. 15

7 Record response, the object type that gets returned when a customer requests a specific record by ID. 15

8 Coreless record response, the object type that gets returned when a customer uses the query endpoint. 16

9 Coreless record response, the object type that gets returned when a customer uses the query endpoint. 17

10 The contents of the query folder. 17

11 Query request body, an object representing the request body the customer provides when doing a request to the query endpoint. 18

12 Query condition, an object representing a condition that will be used as part of a query to the database. 19

13 The contents of the exceptions folder. 19

14 Example of one of our custom exceptions. This particular one is cast when a user requests a page that is out of bounds. 20

15 The contents of the swagger folder 20

16 The contents of the MongoDB module. 21

17 The contents of the web module. 21

18 The contents of the controller folder. 22

19 The contents of the security folder 22

20 The contents of the service folder 23

21 The contents of the exceptionHandling folder 23

22 The contents of the web module test folder 23

23 Root structure of Demo-application 24

24 Folder structure within Src folder 26

25 State handling for main component in App.tsx 26

26 Routing and contexts in App.tsx 27

27 Private routes only allows users that have logged in to view content. 28

28 Folder structure within components folder 28

29 Folder structure within contexts folder 29

30 Folder structure within pages folder 29

31 Folder structure within resources folder 30

32 Folder structure within service folder 30

33 Folder structure within tests folder 30

34 A heavily simplified class diagram of the main flow through the API. Meant primarily to demonstrate the thought process behind the structure rather than being strictly accurate. 32

35 IntelliJ-generated class diagram of the whole API project structure without files used for testing. 33

36 IntelliJ-generated class diagram of the whole API project structure including files used for testing. 34

37 Activity Diagram for user navigation in the Demo Application 35

38 Diagram showing our database structure 36

39 An example of a Record object. 37

40 In this example we use the query function to find all records belonging to someone who has changed their name. The ”systemMeta” field has been omitted in this example. 39

41 Scripts can easily access localStorage's items and do far worse damage than illustrated in this figure. 42

42 Continuous Integration stages 52

43 The Dockerfile used for the deployment stage. 53

44 The deploy stage we initially wrote for Continous Deployment 53

45 Continuous Integration stages for Demo-application 54

46 The Dockerfile used for the deployment stage. 55

47 JaCoCo test coverage of the API. 56

48 Coverage report generated when running npm test 57

1 Introduction

This document is written for the bachelor project in the computer engineering programme (ITHINGDA) at the Norwegian University of Science and Technology. Our bachelor thesis concerns the development of a system called ”Digital Evidence Management” on behalf of our client Signicat. This document serves as an attachment to our main thesis.

What follows is a description of the structure and function of Digital Evidence Management (also refereed to as just ”DEM”). The description is meant to be comprehensive enough so that a developer, new to the project, can understand the inner workings of the system and be able to develop it further. We will cover both the DEM API and Demo Application in this documentation.

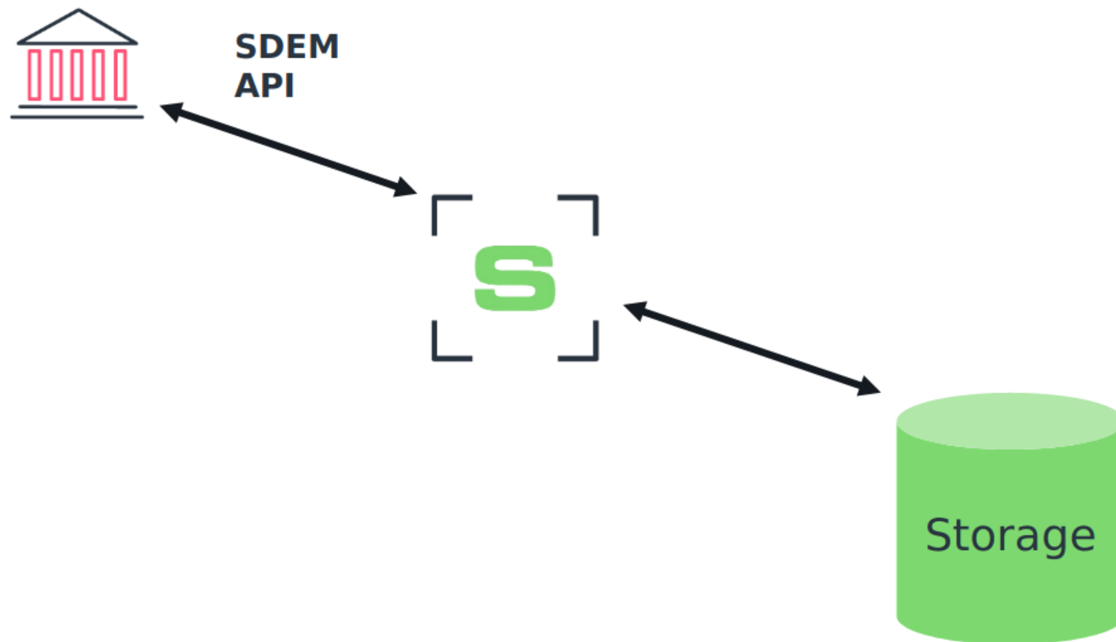


Figure 1: *High level figure of the architecture.*

2 Architecture

The main architecture of Digital Evidence Management is implemented in a three-tier client/server architecture. A customer of Signicat, for example a bank, makes a request to the Signicat Digital Evidence Management (SDEM) API solution. The API processes the request and interacts with the database on behalf of the customer. The customer can connect to the API directly or via the Demo Application, which serves as an UI abstraction.

Our system currently uses the Spring framework for the API and MongoDB as a storage solution. The system has been designed to be modular so that these components can be easily replaced if desired.

3 Project structure

This chapter will describe the structure of our project. DEM consists of two parts: the API (backend) and the Demo Application (frontend). The project follows a MVC (Model View Controller) design pattern where the Model and Controller is covered by the API and View is covered by the Demo Application.

3.1 Backend

3.1.1 Root structure

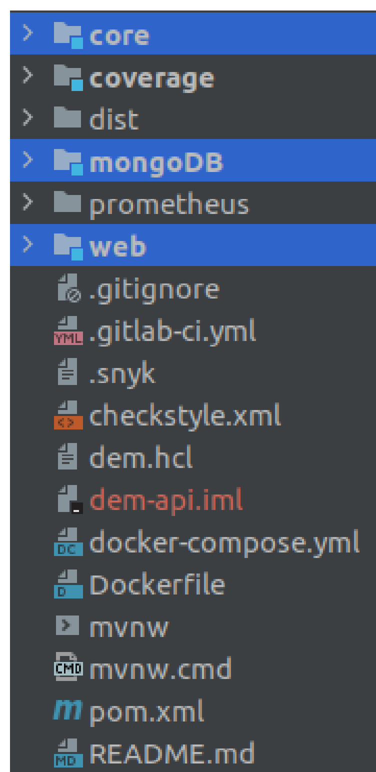


Figure 2: *The root structure for the DEM API.*

The root structure of the API is divided into three main Maven modules: Core, MongoDB, and Web. The fourth module, Coverage, is used for generating code coverage reports and is not essential to the running of the API.

An explanation of the non-module files and folders within root:

- **dist**: A folder for storing OpenAPI specifications that are generated at runtime.
- **prometheus**: A folder containing a .yml file for creating the Prometheus Docker image and a .yml file for creating alerts used by Prometheus.

- **.gitignore**: A rule file for the API repository, where we can specifically tell Git which files to not track.
- **.gitlab-ci.yml**: A template file for Gitlab's CI enviroment. Defines stages for the Continous Integration and Continous Deployment jobs that run when pushing and merging branches in the remote repository on Gitlab.
- **.snyk**: A local policy file to use with Snyk for the API project. Snyk is a tool which checks for vulnerable dependencies. If snyk reports any issues, and no patch is available at the time, the vulnerability can be added to the ignore section in this file.
- **checkstyle.xml**: File setting the Checkstyle rules for the project. Checkstyle is a tool for enforcing rules for code standards. This helps with keeping our code clean and readable. A rule can for example be: a method can not be longer than 140 lines.
- **dem.hcl**: A HCL job file for specifying how jobs and tasks should be run in Nomad on AWS.
- **docker-compose.yml**: File describing to docker-compose how to run our local DB, our local test DB and Prometheus in separate Docker Containers. Also describes what parameters, such as DB usernames and passwords, to feed to the containers.
- **Dockerfile**: Used for describing how a Docker container with the DEM API should be set up.
- **mvnw & mvnw.cmd**: Files allowing Maven to run even if Maven is not installed on the system running the API.
- **pom.xml**: File that contains information about the project, configuration details, and dependencies used by Maven to build the project.
- **README.md**: A text file that follows project with synopsis and instructions on how to run the project and deploying to pre-production.

3.1.2 Core

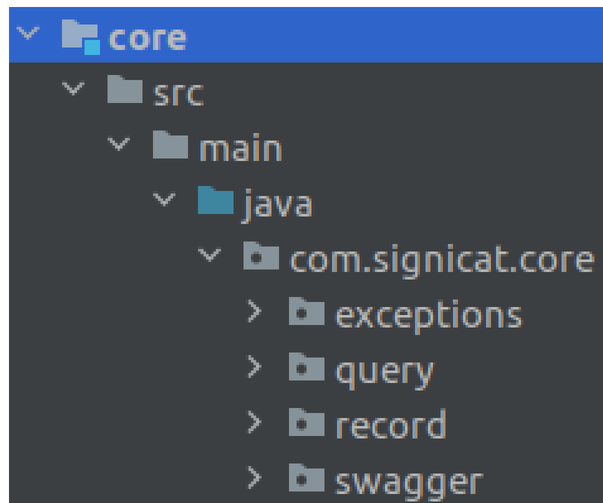


Figure 3: *The contents of the core module*

The Core module covers the Model part of the MVC design pattern. All data classes such as records, immutable POJOs, and Enums are found in this module. We have divided these files into 4 categories: record, query, exceptions, and swagger.

3.1.2.1 Record folder

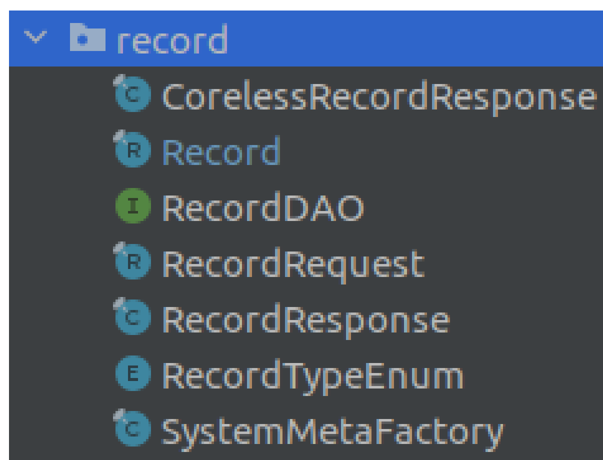


Figure 4: *The contents of the record folder*

The 'record' folder contains our different data transfer objects (except for two in the query folder) and other assisting classes. *Record*, *RecordRequest*, *RecordResponse*, and *CorelessRecordResponse* are all data transfer objects.

Clarification: the record classes (not to be confused with the java 'record' data type) are our representation of the data that is stored in the database. The different record classes reflect how the data is shaped on it's way to and from the database.

RecordDAO is an interface describing the different functions that are required in the DAO implementation.
RecordTypeEnum is an enum containing the allowed record types.
SystemMetaFactory is a class used for generating the system metadata for a record. The system metadata is the data that Signicat attaches to each stored record.

Record

```
public record Record(
        @Id
        @JsonProperty("uuid") UUID uuid,
        @JsonProperty("type") RecordTypeEnum type,
        @JsonProperty("systemMeta") Map<String, Object> systemMeta,
        @JsonProperty("customerMeta") Map<String, Object> customerMeta,
        @JsonProperty("coreData") Map<String, Object> coreData)
        implements Serializable {
}
```

Figure 5: *Record, the data structure that gets stored in the DB.*

The record class is the object type that gets stored in the database. It consists of 5 parameters:

- **UUID**: The uuid is the identifying parameter following the UUID standard.
- **Type**: The type parameter is an enum from *RecordTypeEnum*. It can be used for quick sorting of records.
- **systemMeta**: The systemMeta parameter is the data that Signicat attaches to each stored record.
- **customerMeta**: This is the searchable heterogeneous metadata provided by the user.
- **coreData**: This is the heterogeneous data provided by the user that gets timestamped by the Signicat QTSA.

RecordRequest

```
public record RecordRequest(

        @JsonProperty("type") String type,
        @JsonProperty("customerMeta") Map<String, Object> customerMeta,
        @JsonProperty("coreData") Map<String, Object> coreData

) {
}
```

Figure 6: *Record request, this is the data the customer provides when requesting to post a new record to DEM.*

The record request class is the object that the body of the "post new record" request gets deserialized to. It contains 3 parameters:

- **Type**: The type parameter is a string that later will be checked against *RecordTypeEnum*.
- **customerMeta**: This is the searchable heterogeneous metadata provided by the user.
- **coreData**: This is the heterogeneous data provided by the user that gets timestamped by the Signicat QTSA.

RecordResponse

```
public final class RecordResponse extends RepresentationModel<RecordResponse> {

    @Id
    @JsonProperty("uuid")
    private final UUID uuid;
    @JsonProperty("type")
    private final RecordTypeEnum type;
    @JsonProperty("systemMeta")
    private final Map<String, Object> systemMeta;
    @JsonProperty("customerMeta")
    private final Map<String, Object> customerMeta;
    @JsonProperty("coreData")
    private final Map<String, Object> coreData;
```

Figure 7: *Record response, the object type that gets returned when a customer requests a specific record by ID.*

The record response class is a POJO that gets returned when a customer requests a record from

DEM by ID. Before a record response is returned, it will have it's coreData validated and HATOAS links will be added. The class consists of 5 parameters:

- **UUID**: The uuid is the identifying parameter following the UUID standard. This parameter was used to fetch the record from the database.
- **Type**: The type parameter is an enum from *RecordTypeEnum*. It can be used for quick sorting of records.
- **systemMeta**: The systemMeta parameter is the data that was attached to the record by Signicat before the record was stored.
- **customerMeta**: This is the searchable heterogeneous metadata provided by the user.
- **coreData**: This is the heterogeneous data that was provided by the user when the record was first posted. The coreData gets verified using the QTSA timestamp.

CorelessRecordResponse

```
public final class CorelessRecordResponse extends RepresentationModel<CorelessRecordResponse> {

    @Id
    @JsonProperty("uuid")
    private final UUID uuid;
    @JsonProperty("type")
    private final RecordTypeEnum type;
    @JsonProperty("systemMeta")
    private final Map<String, Object> systemMeta;
    @JsonProperty("customerMeta")
    private final Map<String, Object> customerMeta;
```

Figure 8: *Coreless record response, the object type that gets returned when a customer uses the query endpoint.*

The coreless record response class is a POJO that gets returned as part of a return page when a customer uses the query endpoint in the API. It's important to note that this class does not contain a coreData parameter. This also means that the object is not validated before being returned. If the customer wants a record's core data: they need to request that record by it's ID. The class consists of 5 parameters:

- **UUID**: The uuid is the identifying parameter following the UUID standard. This parameter was used to fetch the record from the database.
- **Type**: The type parameter is an enum from *RecordTypeEnum*. It can be used for quick sorting of records.

- **systemMeta**: The systemMeta parameter is the data that was attached to the record by Signicat before the record was stored.
- **customerMeta**: This is the searchable heterogeneous metadata provided by the user.

RecordTypeEnum

The recordTypeEnum is an enum that can be used for general categorization of the records that are stored in the DB. The enum describes what type of record a record is. This can for example be: *transaction*, *signature*, *log_in*, or *other*. The record types are not intended to limit the type of data that can be stored via DEM, but rather to help the end-user in categorizing their records.

SystemMetaFactory

```
public final class SystemMetaFactory {

    public static Map<String, Object> generateSystemMeta(final String timestamp, final Instant createdDateTime) {
        final Map<String, Object> returnMap = new HashMap<>();
        returnMap.put("timestamp", timestamp);
        returnMap.put("timestampValid", true);
        Instant ttl = createdDateTime.plus( amountToAdd: 1095, ChronoUnit.DAYS);
        returnMap.put("ttl", ttl);
        returnMap.put("createdDateTime", createdDateTime);
        returnMap.put("markedForDeletion", false);
        return returnMap;
    }
```

Figure 9: *Coreless record response, the object type that gets returned when a customer uses the query endpoint.*

The SystemMetaFactory class is used to generate Signicat's metadata. This data is attached to each record before being stored in the database. The system metadata is also where the timestamp and the time to live is stored.

3.1.2.2 Query folder

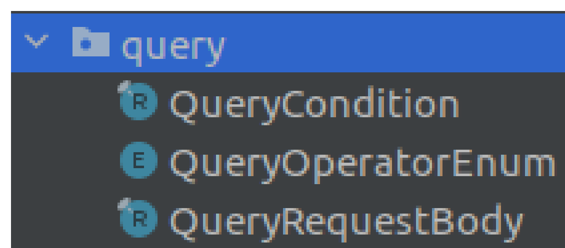


Figure 10: *The contents of the query folder.*

The 'query' folder contains data transfer objects and an enum that are specific to the query endpoint and execution path. *QueryRequestBody* and *QueryCondition* are the DTOs.

QueryRequestBody

```
public record QueryRequestBody(

        @JsonProperty("and") List<QueryCondition> and,
        @JsonProperty("or") List<QueryCondition> or/*,
        @JsonProperty("not") List<QueryCondition> not
        */
)
{}
```

Figure 11: *Query request body, an object representing the request body the customer provides when doing a request to the query endpoint.*

The query request body consists of two lists of *QueryCondition* objects. These two lists are used to form the search query that is used during the query endpoint execution path.

- **And list**: The query conditions in this list will be joined together as part of a query with the logical 'AND' operator. This means that a record in the DB needs to pass every condition in this list in order to be returned.
- **Or list**: The conditions in the 'or' list will be joined together with the logical 'OR' operator. This means that a record in the DB needs to pass at least one of the conditions in this list in order to be returned

There is a possibility to add a third list, 'not', but that is not implemented at the time of writing.

Query condition

```
public record QueryCondition(

        @Parameter(description = "This is a field")
        @JsonProperty("field") String field,
        @JsonProperty("operator") String operator,
        @JsonProperty("value") Object value
)
{}
```

Figure 12: *Query condition, an object representing a condition that will be used as part of a query to the database.*

The Query condition class represent a condition used as a part of a query. You can view these conditions as filters, a record is only returned when querying if the record matches the condition (depending on what list the condition is part of). The *QueryCondition* object consists of 3 parameters:

- **field**: The 'field' parameter denotes which field in the database the condition should be applied to.
- **operator**: The 'operator' parameter marks which query operator should be used to match the value of the database field with the condition value.
- **value**: The 'value' parameter is the value that the contents of the chosen database field will be matched against.

3.1.2.3 Exceptions folder

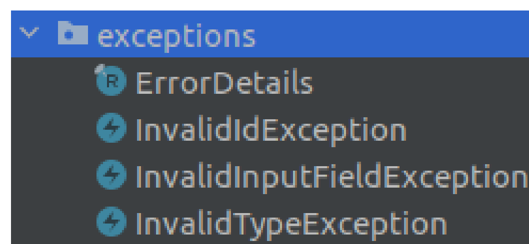


Figure 13: *The contents of the exceptions folder.*

The exceptions folder contains all of our custom exceptions as well as *ErrorDetails* which is a class used to template the response body when one of our custom exceptions is cast. Our custom exceptions are for handling erroneous user actions specific to our API.

```java
public class PageOutOfBoundsException extends Exception {

    final String message = "The requested page is out of bounds.";
    final String details;

    @Override
    public String getMessage() { return message; }

    public String getDetails() { return details; }

    public PageOutOfBoundsException(final String details) { this.details = details; }
}
```

Figure 14: *Example of one of our custom exceptions. This particular one is cast when a user requests a page that is out of bounds.*

3.1.2.4 Swagger folder

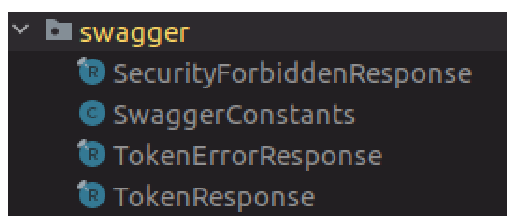


Figure 15: *The contents of the swagger folder*

The files in the swagger folder are used as example objects that are not automatically generated by Spring Doc. Spring Doc is the library we use to help us generate a OpenAPI definition for our API during runtime.

In addition to the model responses there is also a class named ”SwaggerConstants”. Within this class there are constants used for API documentation. The constants in this class ensure that the API documentation will have uniform responses, response statuses, error messages, etc.

3.1.3 MongoDB

The MongoDB module cover all the files that are unique to our chosen database implementation. In our case: MongoDB. These are the files that would need to be replaced entirely when switching database solution.

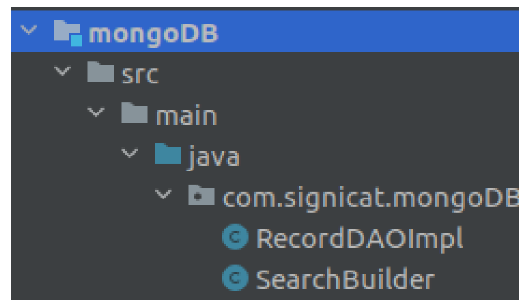


Figure 16: *The contents of the MongoDB module.*

3.1.3.1 RecordDAOImpl

This class deals with the direct communication to our Mongo database. It has one method corresponding to each endpoint in the API as well as other helper methods.

3.1.3.2 SearchBuilder

The *SearchBuilder* class is the class responsible for handling the request body from the query endpoint. It takes the conditions from the body and uses them to build a valid query for the specific database type that is in use. In this case; MongoDB.

3.1.4 Web

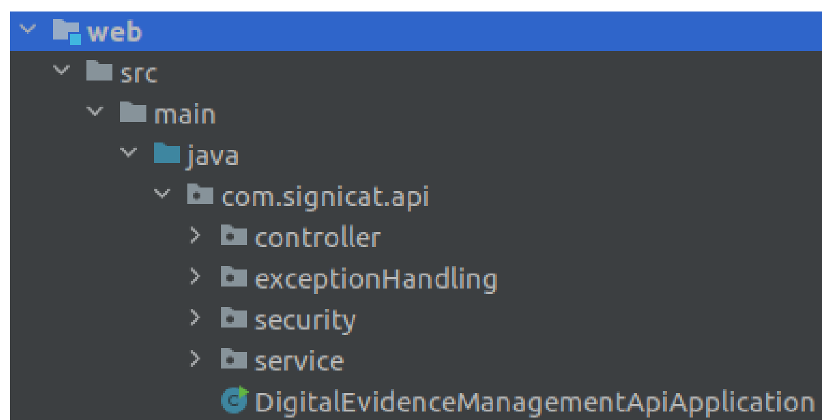


Figure 17: *The contents of the web module.*

The web module contains most of the logic of the application. The controller and service layers are the major elements here. This is also where the main class: *"DigitalEvidenceManagementApiApplication"*, is located.

3.1.4.1 Controller

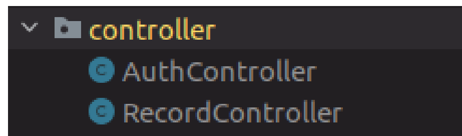


Figure 18: *The contents of the controller folder.*

AuthController

AuthController is a controller with the single endpoint: "Auth". This endpoint is just a proxy endpoint that makes requests against Signicat's own OAuth services. The thought behind this proxy endpoint is to give the customer a single point of entry, so that they would not need to look up documentation on another page and have to deal with separate URLs and services. The Proxy endpoint also gives us more control in regard to CORS configuration.

The auth endpoint can be found at route: "/dem/auth".

RecordController

RecordController is the most significant controller for the API and handles all endpoints that have the route "/dem/records". RecordController handles routing, requests and responses for all record related methods. The logic used for the endpoints can be found in the RecordService class which have methods for storing, retrieving, updating and querying the database.

3.1.4.2 Security

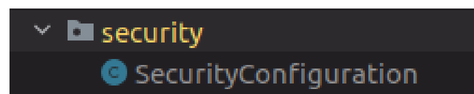


Figure 19: *The contents of the security folder*

SecurityConfiguration

SecurityConfiguration is a class for configuring Spring Security to decode JWT's and to handle authorization and authentication for the API.

3.1.4.3 Service

RecordService

This class contains most of the business logic of the application. RecordService is a store front class based on the *"Facade design pattern"* and provides abstracted methods to communicate with the database. This is where "UUID" and "systemMeta" is added and the "coreData" gets timestamped for new records. This is also where we handle the translation between the different Record types.

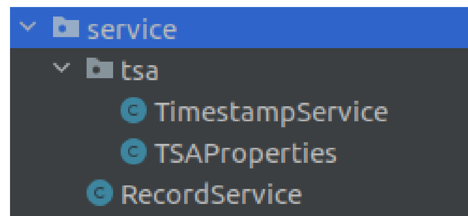


Figure 20: *The contents of the service folder*

TimestampService

TimestampService handles the logic around timestamping data and verifying that timestamped data is still valid. This service is used by RecordService to keep ”coreData” secure.

ExceptionHandling

Figure 21: *The contents of the exceptionHandling folder*

The *ExceptionHandling* class intercepts our custom exceptions if they are cast during runtime. It returns the appropriate HTTP response code according to the exception, along with a message explaining what went wrong.

3.1.4.4 Tests

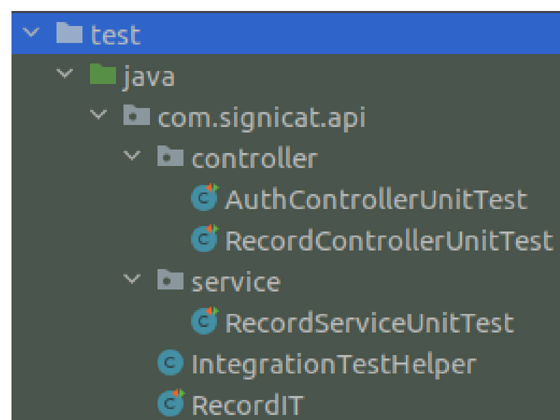


Figure 22: *The contents of the web module test folder*

This is where the unit and integration tests for the services and controllers are located.

3.2 Frontend

3.2.1 Root structure

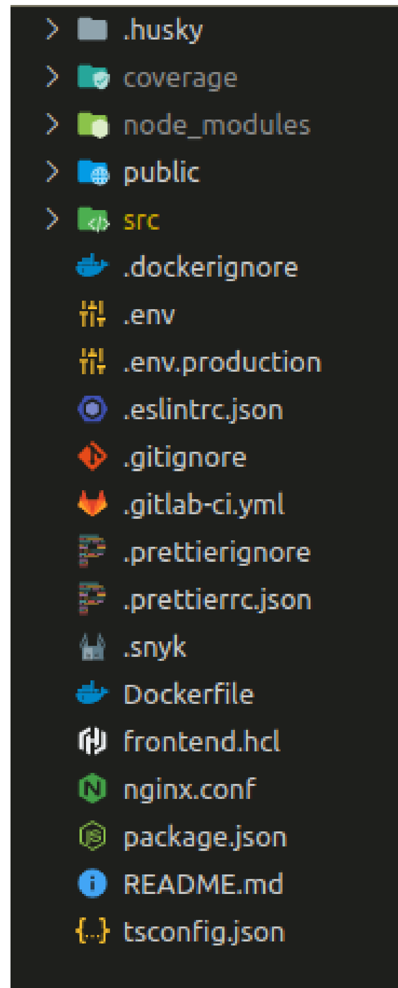


Figure 23: *Root structure of Demo-application*

The root structure has the standard folders: node_modules, public and a src, which is default for any React project. In addition to the standard react boilerplate folders, our root structure also has the folder *".husky"* and the folder *"coverage"*. *".husky"* is a hidden folder containing Git hooks, and the folder *"coverage"* contains auto generated coverage report from running unit tests.

In addition to the folders we have a bit of config files for various needs concerning CI/CD and development.

Explanation of the files within root:

- **Dockerfile:** Template and instruction on how to make a image of a build version of the demo-application.

- **Dockerignore**: There are several dependencies needed for building a docker image of the Demo Application, and instead of writing each file that needed to be copied into the docker container, we found it easier to just use a dockerignore file to make docker ignore some folders and files when copying the needed resources.
- **.env and .env.production:** Environment files for local development and for deployment to pre-production servers.
- **.eslintrc.json:** A local configuration file for Eslint that follows the project. Instead of using global settings that can vary from project to project, we specified in this file which linting rules to follow for the Demo Application.
- **.gitignore:** A rule file for the repository, where we can specifically tell Git which files not to track.
- **.gitlab-ci.yml:** A template file for Gitlab's CI enviroment. Defines stages for the Continuous Integration and Continuous deployment jobs to run when pushing to branch, and/or merging branches on remote repository on Gitlab.
- **.prettierignore:** Rule file for prettier on which folders to ignore when formatting and reporting style errors.
- **.prettierrc.json**: A local rule file for prettier on how to format code based on style preference. Having this local configuration file within the project makes sure that all VS Code extensions will use the local rule set instead of global rules.
- **.snyk**: A local policy file to use with Snyk for the demo-application project. Snyk is a tool which checks for vulnerable dependencies and paths. If Snyk reports any issues, and no patch is available at the time of reported vulnerability, the vulnerability can be added to the ignore section in this file.
- **frontend.hcl:** A HCL job file for specifying how jobs and tasks should be run in Nomad on AWS.
- **nginx.conf:** Configuration for reverse proxy Nginx running inside Docker containers on how to serve the production build of the website.
- **package.json:** A file noting all the dependencies the demo-application has, also has metadata about project, versioning details and scripts.
- **README.md:** A text file that follows project with synopsis and instructions on how to run the project and deploying to pre-production.
- **tsconfig.json:** Compiler options and specification of root files in order to compile Typescript into browser friendly JavaScript.

3.2.2 Src

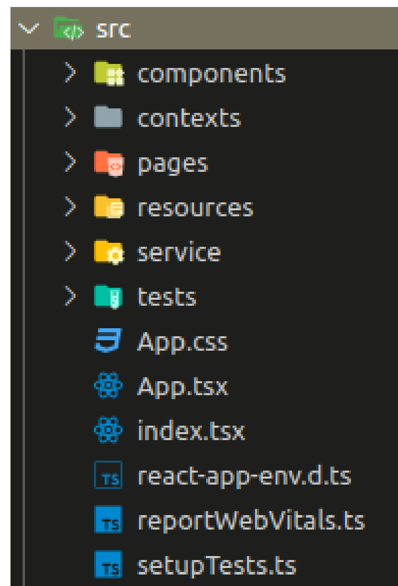


Figure 24: *Folder structure within Src folder*

The entrypoint of our React application is index.tsx which renders the main component App from App.tsx into the div element with a class name: ”root” within the DOM. App is the root parent of all components, and is responsible for routing between pages with React Router, and also has state management in relation to the Context API provided from React Context. App.tsx is also responsible for attaching modals to the DOM to be rendered, and can be called from any sub components. App.css has some global style preferences that concerns the entire project, ie. icons from Material Icons etc.

Since App.tsx is the main parent component that handles context and the routing we would like to highlight how the main structure of the code is.

```
function App() {
    const [alertTitle, setAlertTitle] = useState("")
    const [alertMessage, setAlertMessage] = useState("")
    const [modalVisible, setShowModal] = useState(false)
    const [showAlertModal, setShowAlertModal] = useState(false)
    const [modalContent, setModalContent] = useState<JSX.Element | null>()
    const [alertInputModalVisibility, setShowAlertInputModal] = useState(false)
    const [basicAuth, setBasicAuth] = useState("")
    const [loggedIn, setLoggedIn] = useState(false)
    const [token, setToken] = useState("")
```

Figure 25: *State handling for main component in App.tsx*
As seen in the figure above we can see that App has several states for controlling modals and setting user and login information in memory.

```
return (
    <CredentialContext.Provider value={credentialsValue}>
        <ModalContext.Provider value={modalValue}>
            <Router>
                <div className="appWrapper">
                    <NavBar
                        logoSrc="https://developer.signicat.com/wp-content/themes/manual-child/images/signicat-logo-white.png"
                        navNames={[
                            "Component Gallery",
                            "Testdata Generator"
                        ]}
                        navLinks={["/componentGallery", "/testData"]}
                    />
                    <div className="contentWrapper">
                        <Switch>
                            <PrivateRoute exact path="/">
                                <MainPage />
                            </PrivateRoute>

                            <PrivateRoute exact path="/componentGallery">
                                <ComponentGallery />
                            </PrivateRoute>

                            <PrivateRoute exact path="/testData">
                                <TestDataGenerator />
                            </PrivateRoute>

                            <Route exact path="/login" component={Login} />
                            <Route component={NotFound} />
                        </Switch>
                    </div>
                    {modalVisible && <Modal>{modalContent}</Modal>}
                    {showAlertModal && <AlertModal />}
                    {alertInputModalVisibility && <AlertInputModal />}
                </div>
            </Router>
        </ModalContext.Provider>
    </CredentialContext.Provider>
)
```

Figure 26: *Routing and contexts in App.tsx*

Top encapsulating parents here are <CredentialContext.Provider> and <ModalContext.Provider> which provides all children access to global states. Also, here one can see that some routes are placed as children to the <PrivateRoutes> component, thus separating routes into public and routes. Private routes would need to be logged in to view the content at a specific private route.

```
function PrivateRoute({ children }: any) {
    const credentials = useContext(CredentialContext)
    return (
        <Route
            render={() =>
                credentials.loggedIn === true ? (
                    children
                ) : (
                    <Redirect path="" to="/login" />
                )
            }
        />
    )
}
```

Figure 27: *Private routes only allows users that have logged in to view content.*
The PrivateRoute component is a child of the CredentialContext and can access the global state using the useContext hook, and uses conditional rendering to display the content based on whether the user is logged in or not. If the user is not logged in, they will be redirected to the login page.

3.2.2.1 Components

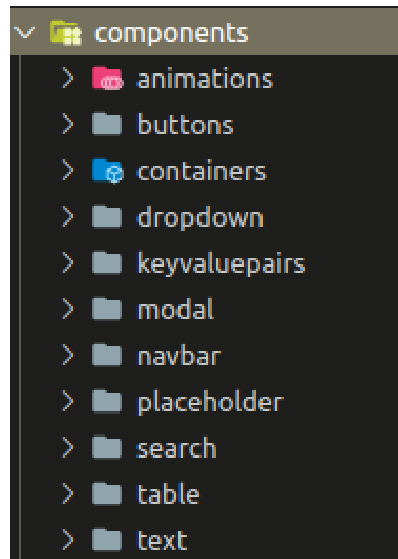


Figure 28: *Folder structure within components folder*

Within the Components folder, we have placed all of the smaller components that are used on the various pages and modal pages in the project. Most of the components within this folder can be re-used as they are written very general with optional props, while some components are too specific

and may need some modifications to be general enough in terms of re-usability.

3.2.2.2 Contexts

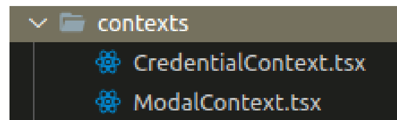


Figure 29: *Folder structure within contexts folder*

The folder Contexts holds all created contexts along with type definitions. These contexts can be called on by either a Provider component or a Consumer component within the project, and will allow for the project to have global state. We have two contexts: CredentialContext and ModalContext. CredentialContext is responsible for storing the global user info and login state, while ModalContext is holding the state and logic of displaying content-modals, alert-modals and input-modals.

Since the context API is similar to Redux state management, it should be easy to move over to Redux if the size and complexity of the project increases.

3.2.2.3 Pages

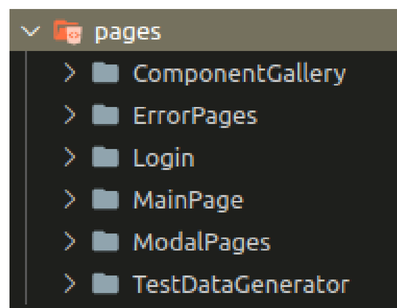


Figure 30: *Folder structure within pages folder*

All pages that are used by React router to switch between pages are found in the pages folder. We can also find a subfolder within this folder named ”ModalPages” where we can find all pages that are rendered inside modals. These pages uses components that are declared within the components folder. Each page has their own folder with at least two files in them. For example, login has the files: Login.tsx and login.modules.css. Login.tsx is responsible for the logic and templating, while login.modules.css is responsible for the styling of the pages.

3.2.2.4 Resources

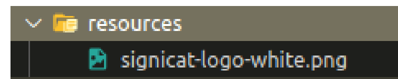


Figure 31: *Folder structure within resources folder*

The resources folder only has a logo inside it, but this is where we would place all assets like media-content, images, fonts and so on. For icons, we are using a content delivery network (CDN) to fetch the icons needed. We could also have stored the icons within the resource folder, but there were no explicit need for it at the time of implementation, and the loading time for the icons used was negligible as we only needed a few icons for this project. If the project should increase in complexity and with a need for more icons, we would consider just storing the icons needed instead to reduce loading time.

3.2.2.5 Service

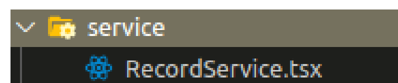


Figure 32: *Folder structure within service folder*

Within the service folder we find the class RecordService in the RecordService.tsx file. This class is a wrapper class with static methods needed for making requests against the Digital Evidence Management API. This service serves as a connecting layer between demo-application and API. RecordService has methods for automatically adding needed headers and parameters, and can query, get record by id, post a record, get used fields in supplied meta data and fetch access tokens, and so on. This class uses the library "Axios" to make HTTP requests. We have also configured Axios interceptor within this file to check the validity of an access token before each request is made, and make a new request to get a new token if the one used is expired before performing the initial request.

3.2.2.6 Tests

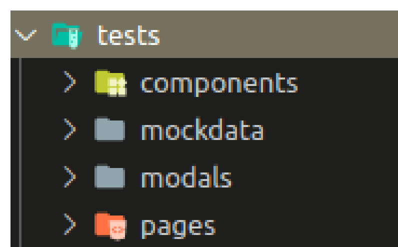


Figure 33: *Folder structure within tests folder*

The tests folder contains all Jest unit tests for the demo-application. The folder structure in tests mirrors how we structured Src. This is done to try to make a clear testing structure that maps to the source files and how we develop components and pages there. We also have the folder mockdata in the tests folder, this is where we placed all data that we mocked to return from the API.

4 Diagrams

4.1 API

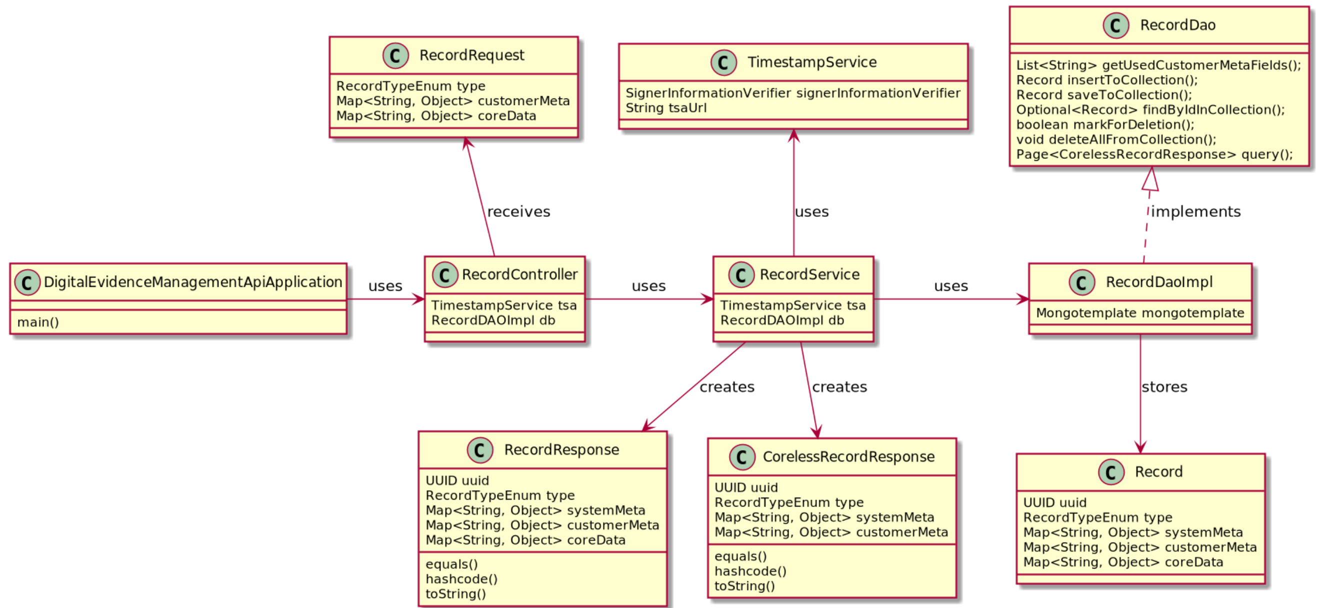


Figure 34: *A heavily simplified class diagram of the main flow through the API. Meant primarily to demonstrate the thought process behind the structure rather than being strictly accurate.*

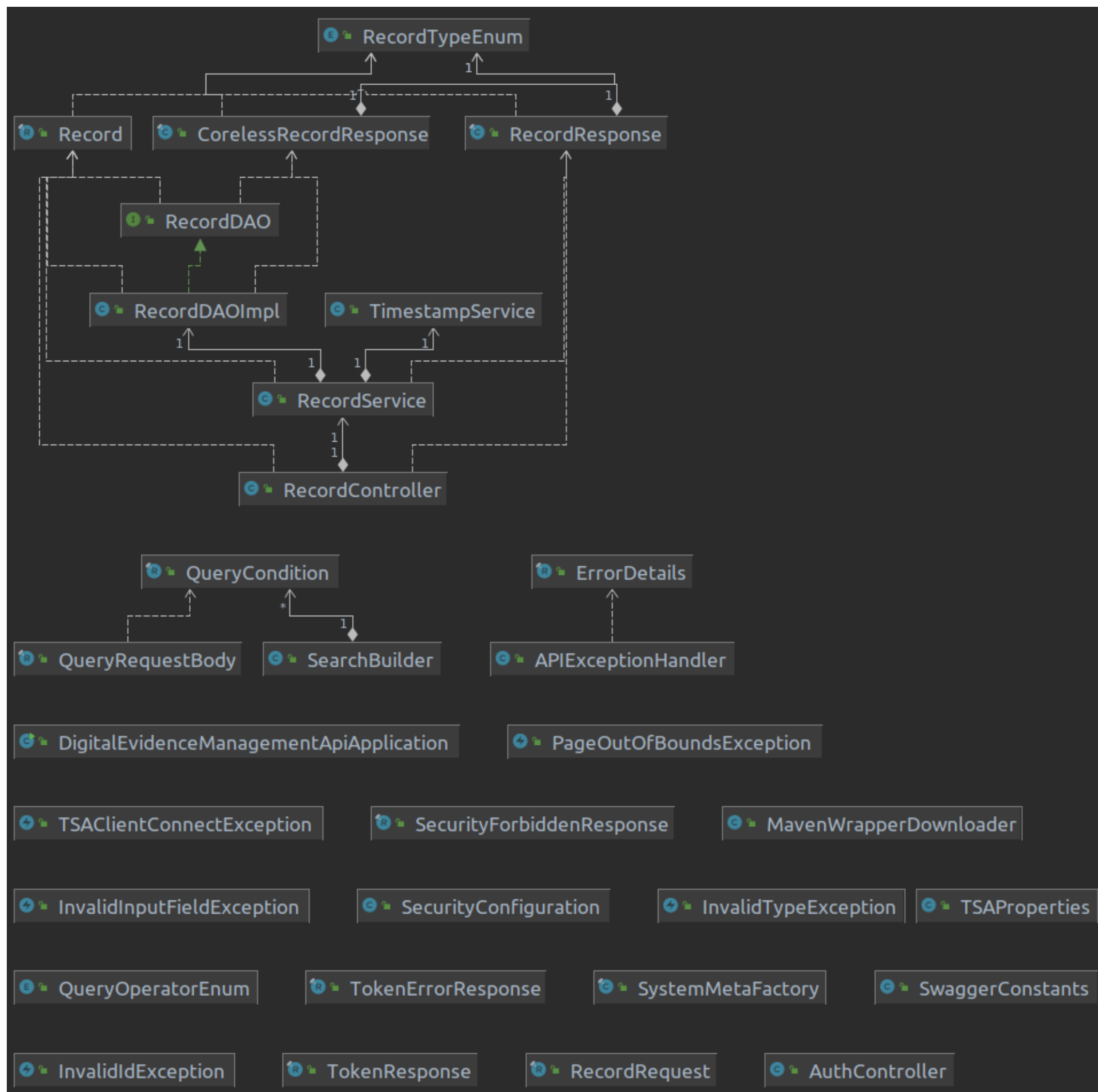


Figure 35: *IntelliJ-generated class diagram of the whole API project structure without files used for testing.*

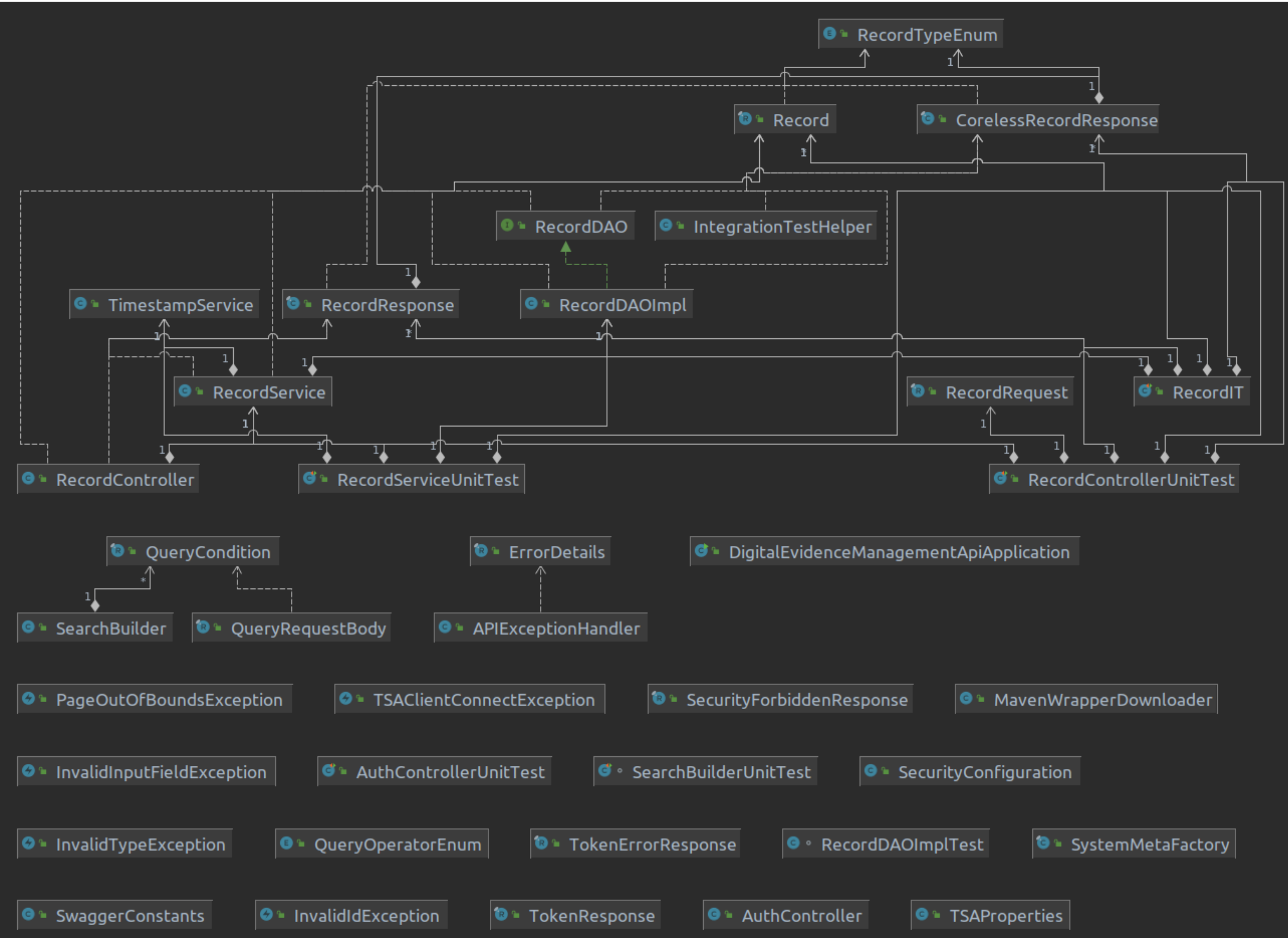


Figure 36: *IntelliJ-generated class diagram of the whole API project structure including files used for testing.*

4.2 Demo Application

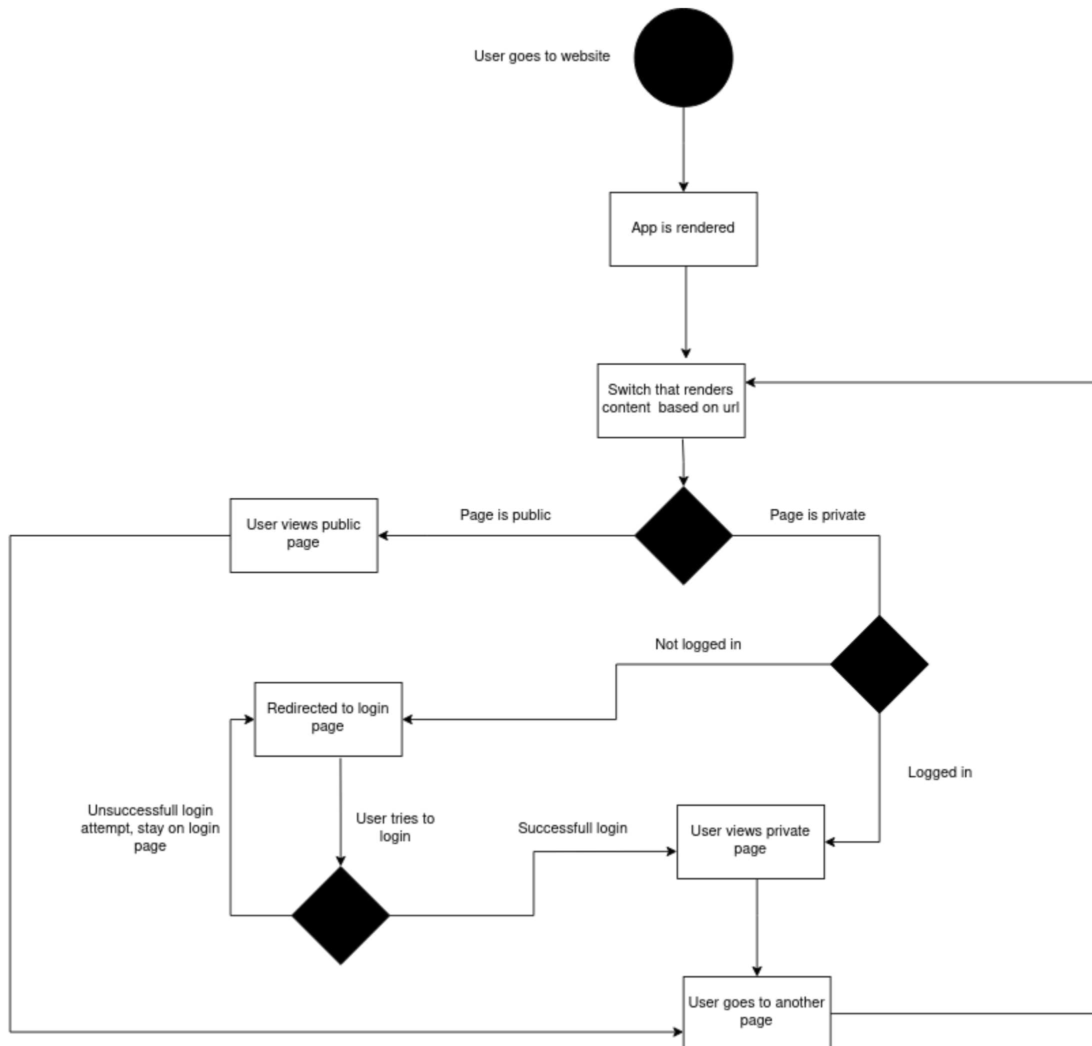


Figure 37: *Activity Diagram for user navigation in the Demo Application*

5 Database model

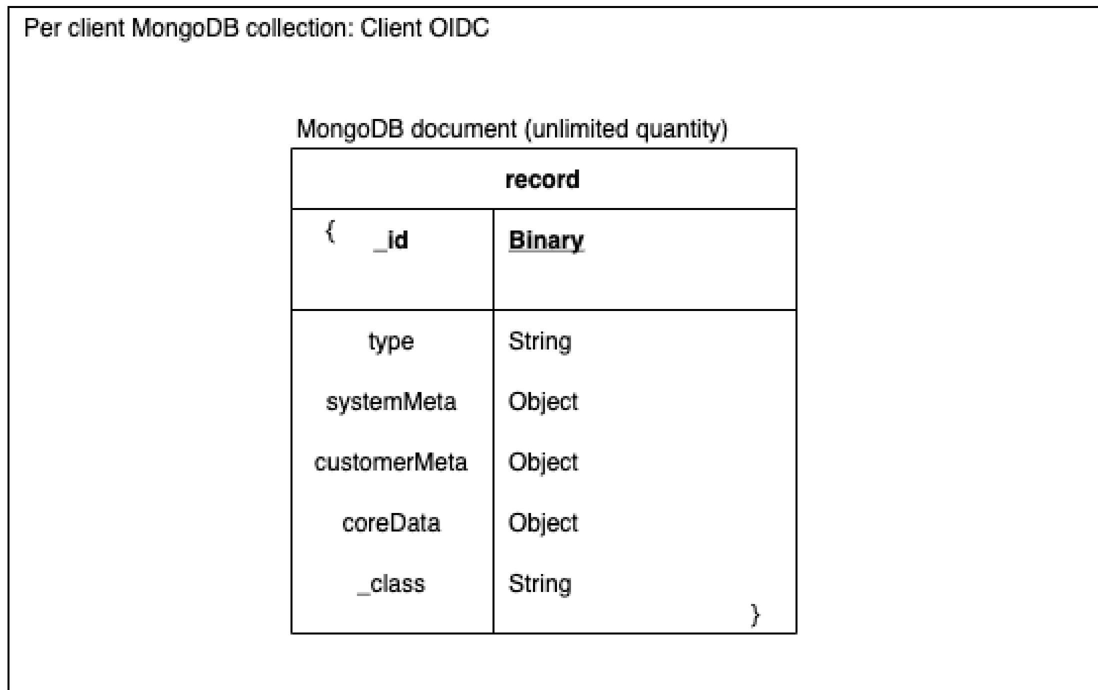


Figure 38: *Diagram showing our database structure*

6 Server services

The system really only focuses on managing one resource, the Record entity. As such the API's main functionality revolve around handling this entity.

The full API Documentation can be found at:
`https://signicat.gitlab.io/signature/digital-evidence-management/`

6.1 Adding a record

/dem/records

By using the POST method on this endpoint it is possible to add a record a to the system. The required fields for this operation is: ”type”, ”customerMeta” and ”coreData”, where ”type” is a String value that must match our predefined types and ”customerMeta” and ”coreData” is open to contain any data specified by the user. A unique identifier and the ”systemMeta” field will be generated by the system for the record added. As the record is stored the ”coreData” is also timestamped so that it can be verified in the future.


```
1  {
2      "uuid": "d10ca55a-b64b-48b8-a5aa-6b1a87bf53ea",
3      "type": "TRANSACTION",
4      "systemMeta": {
5          "createdDate": "2021-05-13T00:00:00Z",
6          "createdDateTime": "2021-05-13T09:00:18Z",
7          "timestampValid": true,
8          "markedForDeletion": false,
9          "ttl": "2024-05-12T09:00:18Z",
10         "timestamp": "MIICoDAVAgEAMBAMDk9wZXJhdGlvbi
               3VMsOLUFnsTIpwIUJVeW2W/E0VCjt1ukOUkoBcZ3
               +xMMA0GCWCGSAFlAwQCAQUAoIIBGzAaBgkqhkiG9
               jtQh1KT4PiHzO4Isqo2plQpudDqcwuQXCb5EwgYE
               +MDwxCzAJBgNVBAYTAk5PMRQwEgYDVQQKEwtTaWd
11     },
12     "customerMeta": {
13         "firstName": "Olav",
14         "lastNames": "Andersen",
15         "issuer": "Komplett bank",
16         "dateOfBirth": "1971-07-03T00:00:00Z",
17         "securityLevel": 2,
18         "phoneNr": "47962392",
19         "orderDate": "2021-04-03T00:00:00Z",
20         "relatedRecords": [
21             "5e8d92f3-eff6-4842-b546-73cbded02ecd",
22             "b1fddc8c-936e-41a1-9a29-270e7dded14b",
23             "0f8feb57-c269-4d73-8f2a-f814134219a8"
24         ],
25         "extraInfo": null
26     },
27     "coreData": {
28         "transactionID": 267452797,
29         "fromAccount": 166677944,
30         "toAccount": 433451068,
31         "kid": 451782564,
32         "status": "COMPLETE"
33     }
34 }
```


Figure 39: *An example of a Record object.*

6.2 Retrieving a record

/dem/records/{id}

It is possible to retrieve any record from the database by using the GET method on this endpoint and adding the id of the record to the URL. In the case of the record shown in Figure 39 this would be: ”/dem/records/d10ca55a-b64b-48b8-a5aa-6b1a87bf53ea”

Included in this endpoint is a check to see if the ”coreData” has been altered since the initial post of the record. The ”timestampValid” value locate in the ”systemMeta” will then be set to reflect the integrity of the data.

6.3 Marking a record for deletion

/dem/records/{id}

Any record entered into the database will be given a Time to live (TTL) of 3 years, after which the database will delete a record. Users are able to influence this by marking records for deletion. This will move the TTL date to 30 days from the current date.

6.4 Retrieving specific records

/dem/records/query

The server also supports a search ability where the user can specify which records to retrieve by creating a detailed query. By using the POST method on this endpoint the user will be able to send in different criteria to get the desired list of records. This can be used to retrieve all records, filter the list of records or search for specific records.

This is done by constructing criteria consisting of a value of some kind and specifying what field is targeted and what operation should be executed. The full list of operators are as follows:

Query Operation	**Description**
eq	Checks if the field's value matches the input value exactly.
ne	Checks if the field's value does not match the input value.
gt	Checks if the field's value is greater than the input value. Works best with numeric values.
gte	Checks if the field's value is greater than or equal to the input value. Works best with numeric values.
lt	Checks if the field's value is less than the input value. Works best with numeric values.
lte	Checks if the field's value is less than or equal to the input value. Works best with numeric values.
regex	Checks if the field's value contains the input value using regex. Works best with String values.
in	Checks if the field's value matches any of the values in the inputted array.
nin	Checks if the field's value does not match any of the values in the inputted array.

These criterias can be put into an ”and” list to signify that all criteria in the list must be fulfilled. Criteria can also be placed in an ”or” list to signify that only one criteria must be fulfilled.

```
1  {
2      "and": [
3          {
4              "field": "customerMeta.firstName",
5              "operator": "eq",
6              "value": "Olav"
7          }
8      ],
9      "or": [
10         {
11             "field": "customerMeta.lastName",
12             "operator": "eq",
13             "value": "Andersen"
14         },
15         {
16             "field": "customerMeta.lastName",
17             "operator": "eq",
18             "value": "Gundersen"
19         }
20     ]
21 }
22
```

(a) *Example of a query request*

```
1  {
2      "content": [
3          {
4              "uuid": "65c7dad4-715a-4bc0-a944-c7ba2dd86303",
5              "type": "TRANSACTION",
6              "systemMeta": {},
7              "customerMeta": {
8                  "firstName": "Olav",
9                  "lastName": "Gundersen"
10             },
11             "links": [
12                 {
13                     "rel": "self",
14                     "href": "http://localhost:9002/dem/records/65c7dad4-715a-4bc0-a944-c7ba2dd86303"
15                 }
16             ]
17         },
18         {
19             "uuid": "287e99fb-1462-4464-aa2a-92249e7062b2",
20             "type": "TRANSACTION",
21             "systemMeta": {},
22             "customerMeta": {
23                 "firstName": "Olav",
24                 "lastName": "Andersen"
25             },
26             "links": [
27                 {
28                     "rel": "self",
29                     "href": "http://localhost:9002/dem/records/287e99fb-1462-4464-aa2a-92249e7062b2"
30                 }
31             ]
32         }
33     ],
```

(b) *Example of a query response*

Figure 40: *In this example we use the query function to find all records belonging to someone who has changed their name. The "systemMeta" field has been omitted in this example.*

6.5 Others

/dem/v3/api-docs

- Returns an OpenAPI specification file

dem/swagger-ui.html

- Opens a UI containing the OpenApi documentation.

dem/actuator/health

- Returns application health information

dem/actuator/prometheus

- Exposes metrics in a format that can be scraped by a Prometheus server.

dem/auth

- Proxy endpoint that let customers make request to obtain access tokens from Signicat's OAuth services through DEM's API.

7 Security

With respect to the top 10 OWASP standard security awareness document, we have identified that we have some security vulnerabilities, but we have also taken steps to address certain vulnerabilities [1].

A1:2017 Injection
Since we are using MongoDB, a NoSQL database solution, the database is not vulnerable to any SQL injections. We are also not executing direct queries into MongoDB.

A2:2017 Broken Authentication
The API's session cookie policy for generating session tokens is set to be stateless. There is no distributed session cookie sent back to user, so it not possible to hijack a session cookie as there is none to capture. The credentials like client ID and client Secret is generated by Signicat's own OAuth service. We can not ascertain how secure that system is, but given that Signicat is a trusted business: we have reason to be confident in the system's security.

A3:2017 Sensitive Data Exposure
Both the API and the Demo Application that are deployed on AWS are set up to use encrypted communication through HTTP with TLS certificates (HTTPS). Full in-transit security is not achieved at this point, as there is a error with the certificate name. We have contacted Signicat to fix this, as they are the ones holding the certificate. The current solution is not encrypted at rest, but there have been made preparations for switching from MongoDB Community Edition (no encryption at rest) to MongoDB Enterprise edition which allows for encryption at rest. Switching to the Enterprise edition costs money, so until Signicat has approved this and ordered an instance of the Enterprise to run in AWS: the database is unencrypted. Theoretically it should be as simple as pointing to another MongoDB database and provide relevant credentials for that database.

A4:2017 XML Injection
Our API does not consume XML data, all endpoints only accepts data in the form of "application/json". This vulnerability is not applicable to our system.

A5:2017 Broken Access Control
Digital Evidence Management uses OAuth 2.0 with a client-credentials protocol flow, where we have different scope permissions for every endpoint. We have set up the scopes: "Admin", "Read", "Write", where Oauth clients with read permission are only allowed to make GET requests. POST request require the "Write" scope. Accessing other administration endpoints like metrics endpoint requires users to have the "Admin" scope.

The Demo-application has no direct linked uri to requests, so tampering with url will not give unintended access. An example attack vector of this sort would be "/?user=1", where one could just set "user=2" and gain access to another user's information. We do not have insecure direct

object references through URL, because we do no have object reference as a condition for rendering our content.

Furthermore, the database has the different users data separated into collections (SQL table equivalent). Each collection is linked to a client ID. This client ID is parsed from the users provided access token using OIDC. No user is able to gain access to other users data without know the other user's client ID and client secret, or access token.

The Demo Application has also set up Private routes and public routes, so it will only display information pertaining to a user if the current user has logged in with valid credentials.

CORS origin policy for API is set to only allow from https://dem-aws-frontend.signicat.net

A6:2017 Security Misconfiguration
We have taken steps to secure that exceptions thrown in Digital Evidence Management's API does not contain stack trackes, and instead gives custom error messages in response.

Standard passwords has been changed. We should note that passwords for the databases are stored in *application.properties* at this time, and is even accessible in our Gitlab repository. The passwords and database details in *application.properties* are placeholders however, and will be overwritten when feeding in new *application.properties* that are given during deployment.

A7:2017 Cross-Site Scripting XSS
For the Demo Application, there are some serious note worthy security faults within the system. The Demo Application stores both a basic string, which is a Base64 encoding of "clientID:clientSecret", and a access token in localStorage. Base64 encoded strings are not encrypted and can easily be decoded again and thus an attacker would then know all of a client's secret credentials and have full access to the system. The access token can also be used to gain temporarily access to the system until the token expires. By storing sensitive data in localStorage, the system is vulnerable to XSS attacks. Although it should be noted that since we are using React.js for developing the demo-application, and not setting inner-html or allowing user defined href's, it is hard to inject scripts because of the way React sanitize and escape all input, and only displays input as text, but it is theoretically possible.

```
>> for(var i=0; i<localStorage.length; i++){
     console.log(localStorage.getItem(localStorage.key(i)))
   }
```

cHJvZC5ERU1fY2xpZW50LjE6TUpkQUVlN3RjYUlHb1BGNX
kzeUZMTFJLOUxDMjJwaldYRUVjTExzOEYzanN1NFRP debugger eval code:2:11

eyJhbGciOiJSUzI1NiIsImtpZCI6InNpZ25pbmcta2V5LT
VkNWYyYmZkOTZkMjRlOTJiZDZjODEyNjBiZGRmZGY0Iiwi
dHlwIjoiYXQrand0In0.eyJuYmYiOjE2MjA4MDcwMDEsIm
V4cCI6MTYyMDgwNzMwMSwiaXNzIjoiaHR0cHM6Ly9hdXRo
LnNpZ25pY2F0LmlvIiwiYXVkIjoiYXBpOi8vMzAzOWFmYj
g0MDhiNGRhNmFhMzMwY2Y1MzZhMWZhOTYiLCJjbGllbnRf
aWQiOiJwcm9kLkRFTV9jbGllbnQuMSIsInNpZ25pY2F0X2
FjY291bnRfaWQiOiI1MzVmYzFmYy01YzcxLTQ3NDMtYmYx
Mi1hYWU5MDA5NDVkYTgiLCJqdGkiOiJGNTMzNkNCMjYyRT
Y5MTEwMzQ0QzEzMjBBMDYzRTM0QSIsImlhdCI6MTYyMDgw
NzAwMSwic2NvcGUiOlsiYXBpOi8vMzAzOWFmYjg0MDhiNG
RhNmFhMzMwY2Y1MzZhMWZhOTYvREVNOnJlYWQiLCJhcGk6
Ly8zMDM5YWZiODQwOGI0ZGE2YWEzMzBjZjUzNmExZmE5Ni
9ERU06d3JpdGUiXX0.juvX5EjcPzvXXBCf4cK002ivnMKz
tEpjmfhJS38vjl8bGhzZYwQRrs2vVUMgRlvXLPgd5HXlT8
FEruR3asytAyRXGIzDZhqOg2AyGZGShRBUBjsJ4pqP3OJs
85n781fTu4EsfoMUy1la4NzkNQ4QCNwAtnh7TCwhyYVKT4
v5qqXd-q5LqGI5CPEmYR1kiULcWDojs4KXWz-
ii9KD_NyjVXRefnwU9XRc77fdo0FmUorCdAoWnURuNSPU4
0X7t06ssTAlm1M-
3VpjaVmhGb8s2XLeJn6IHBPyB88HoY1E8y9U56VQiRuHs8
gu2M6k3I-Zyaim0Zk5MYp-Ymo9ijpvEw debugger eval code:2:11

Wed May 12 2021 10:14:52 GMT+0200 (Central European Summer Time) debugger eval code:2:11

Figure 41: *Scripts can easily access localStorage's items and do far worse damage than illustrated in this figure.*

The reasoning behind why we are storing such sensitive content in localstorage is due to the nature of choosing a OAuth 2.0 protocol flow that works best for the API. Since the API was the main product that we developed and is supposed to go into production, we chose to prioritize setting up an authentication flow that was best suited for the API. We could off course set up another login solution, but due to time limitations, we did not have time to set up a login solution that can handle user event driven protocol flow as well as a the client-credentials protocol flow that the API uses.

Since the demo-application together with the API is only intended to be set up in a sandbox environment that we call "pre-prod" where new customers can get accustomed to our system, there were really no risks in regards to leaking the client credentials as the sandbox environment is not intended for storing sensitive information. Storing the client credentials in localstorage was necessary to have persistent log-in that survives page-refreshes.

Before storing the credentials in localstorage, all information was only stored in-memory. Conversations with Signicat lead us to favor a better user experience with persistent log-in rather then a safe solution with only storing the credentials in memory. More on this can be found in the main report.

A8:2017 Insecure Deserialization

It is not ascertained if Digital Evidence Management suffers from this vulnerability. More research into the issue and our current solution is required to reach a conclusion on this. We are not using any serialization and deserialization ourselves, but we know that Spring Boot uses a library called

"Jackson" to serialize and deserialize and convert other datastructures into Java data structures, so it is entirely possible that there might be some vulnerabilities there.

A9:2017-Using Components with Known Vulnerabilities
Both the Demo-application and the API uses snyk to check the projects against known vulnerabilities. Snyk is also integrated into Gitlab's CI testing pipeline and is also run on local builds. By using snyk we are alerted of any vulnerable libraries, components and frameworks, and are provided remedies if there are any available. If there are no remedies to vulnerabilities available, we are able to ignore the vulnerability for a time limit, and when the time limit passes we are alerted again and can check if there are any patches released for the vulnerability.

In the Demo Application, we have written all of the components ourselves, and are not using any libraries for components or any frameworks for styling. This is not to say that we are using other dependencies, we are using some other dependecies i.e "Axios", "Base64", "UUID" and so forth, but we have tried to keep it at a bare minimum of external dependencies, and have also tried using only official and well known repositories from NPM with a good user base and reputation.

We have also made steps to make sure that any development dependencies is in a correct grouping so that we are not adding any unnecessary dependencies to the build of the website. Eslint for example is only required for linting while developing, but is not needed when building a production ready website.

A10:2017 Insufficent Logging Monitoring
We have set up metrics to be gathered and served using Spring Actuator and Micrometer. The metrics endpoints are protected and would need a persmission scope of "DEM:Admin" to access. With metrics implemented, it is possible to monitor if there any attacks to the API, and set up alerts and actions on how to respond to any possible attacks. We have not integrated the API with a centralized logging solution, so there is no record stored of potential attacks.

8 Installation and running

8.1 API

8.1.1 Synopsis

The Digital Evidence Management API is used for storing digital evidence securely based on user defined heterogenous metadata and coredata. In addition to the user defined data, Signicat stores it's own system-metadata along with the user provided data in order to secure and organize each record stored in the database.

Main technology used is Java 15 with preview enabled and MongoDB as a storage solution. It is written with interchangeability in mind so that database solution can be changed to other databases. This can be done by creating a new class that implements the methods specified in the RecordDao interface, and changing the implementation used in the RecordController.

This application uses services like QTSA for evidence integrity and OAuth 2.0 with a OIDC layer for authorization and authentication.

8.1.2 Pre-requisites

First, extract the provided zip file.
Open up the *"digital-evidence-management"* folder in an IDE or editor.

Note: *"Root folder"* as referenced in the steps below is: *"digital-evidence-management"*.

Before building or running the application, make sure to run the database and prometheus.
This can be done by running the command **docker-compose up** in a terminal from root folder.

Since DEM is using preview features of Java 15, it needs to run with preview enabled. This can be done by setting the project language level in the project structure(IntelliJ) or by running the application with the flags:
"--enable-preview"

8.1.3 Error handling for database

Docker compose will run cached containers if there are any existing containers already. Often a clean container will fix problems.

Running clean containers:

1. Run command ”docker-compose down” from root folder. **(Removes all network and cached containers.)**
2. Then run ”docker-compose up” again.

8.1.4 Build project

Run a clean build of project with the command:
mvn clean install

8.1.4.1 Build project without running snyk or tests

Run the command:
”mvn clean install -DskipTests -Dsnyk.skip”

8.1.5 Run project

Run the following command from root folder:
First run: **”mvn compile”**

Then application can be runned with the command from root folder:
”java --enable-preview -jar ./web/target/web-1.0.0-SNAPSHOT.jar”

We used IntelliJ to start the project, so there were not much effort put into running the application from the command line. To run the project in IntelliJ click the *”green run”* button, or *”Shift+F10”*.

Note: When running project locally, spring profile defaults to ”dev”.
See Profiles section for more information.

8.1.6 API-Documentation

8.1.6.1 SWAGGER/OPENAPI docs

During runtime of the API, both OpenAPI spec file (oas), as well as static swagger html page is genererated and served automatically by the use of the ”SpringDoc” plugin.

- Swagger page can be found here: `http://localhost:9002/dem/swagger-ui.html`
- OAS spec file is served here: `http://localhost:9002/dem/v3/api-docs`

8.1.6.2 Redoc

Last stage of CI on master branch will also publish updated API documentation in a Redoc format and publish to Gitlab pages.

Redoc API documentation can be found here:
`https://signicat.gitlab.io/signature/digital-evidence-management/`

8.1.7 Metrics

Prometheus server starts on docker-compose and runs on http://localhost:9090.

DEM's API is serving metrics to be scraped by Prometheus on endpoints such as:

- /dem/actuator/prometheus
- /dem/actuator/metrics
- /dem/actuator/health
- /dem/actuator/up

Config and alerts are set in prometheus from the folder ”prometheus.”

Prometheus dashboard can be accessed from:
`http://localhost:9090/`

8.1.8 Profiles

There are currently set up these profiles for the project:

- dev (for running locally)
- preprod (Settings used in AWS pre-prod environment)
- prod (Not deployed to production at this time)

If no profile is selected, spring application will use application.properties by default, which points to the dev profile.

To change between profiles, run with argument flag:
”–spring.profiles.active=<profile>”

8.1.9 Switching databases

The project currently consists of three modules which are dependent of each other in the following order:

- Core: Contains models and core elements needed in further modules.
- MongoDB: Contains the database access object implementation and MongoDB specific code.
- web: Contains the controllers, services and configuration of the API.

The database can be changed by replacing the 'MongoDB' module. The web module will need to have a Maven Dependency to the new module.

All required database functions are defined in the interface 'RecordDAO' in the 'core' module and must be implemented in the new database module. The new module should implement this interface to create a new DAO. The service classes in the 'web' module will also need to import the new Dao object in the same way it currently imports 'RecordDAOImpl' from the 'MongoDB' module.

You would also need to add relevant properties to correct application.properties profiles in 'web' module, i.e datasource and etc. Note: You have to remove MongoDB properties, as the applicaton will crash after trying to connect to MongoDB 3 times and fail.

8.1.10 Static code analysis tool: Checkstyle

Checkstyle is a static code analyser tool that can be found within the project. Checkstyle is used for analyzing the code for style errors, programming errors and bugs.

In root of project structure there is a config file named "checkstyle.xml" which is configured for use with the checkstyle dependecy within the project. The rules defines styling guide and is used to help avoid common programming errors and reduce the number of bugs within the project.

8.1.11 Dockerfile

Within the project there is a Dockerfile templating how a container should be built of this project. This Dockerfile is used on Gitlab's CI stage: Deploy, which is only triggered on the master branch. The Dockerfile consists of a single stage, packaging and compiling down the API into a Jar file, which is then copied over to docker container and is set to run the jar file on docker container execution and expose API's port number.

8.2 Demo-application

8.2.1 Synopsis

A frontend Demo Application for DEM-API written in Typescript using React.js. The Demo Application's function is to showcase the usage and capabilities of DEM's API, and serve as a onboarding solution for getting customers acquainted with the Digital Evidence Management system.

The DEM API can be found at:
`https://gitlab.com/signicat/signature/digital-evidence-management`

8.2.2 Pre-requisites

First, extract the provided zip file.
Open up the"digital-evidence-management"folder in an IDE or editor.

Note:"Root folder"as referenced in the steps below is:"demo-application".

Dependencies need to be installed before running the application locally.

Install all dependencies as stated in package.json file with the command:
npm install

Note: You will also need to run both the API and MongoDB as stated in API installation instructions to have full functionality on the Demo Application.

8.2.3 Run application

Start up the frontend application in devopment mode with command:
npm start

Frontend application can then be found on:
`http://localhost:3000`

Note: The page will reload if you make edits to source files, and you will also see any lint errors in the console if there are any.

8.2.4 Run tests

To run all tests:
npm test

To test a single class:
npm test <THE_COMPONENT_YOU_WANT_TO_TEST>
For example: **npm test Table**

Tests are written inside their own file and have a naming structure i.e ”Table.test.tsx”. In this example we have stored all tests relating to a component called Table in that file.

8.2.4.1 Test coverage report

Running **npm test** will generate a coverage report.

Coverage report can be found in the folder coverage in root folder.

Open up `/coverage/lcov-report/index.html` in a browser to see coverage report.

8.2.5 Build production app

To build a production ready static hmtl page, run command:
npm run build

This command builds the app for production and places it in the ‘build‘ folder. When running the build command, build will use variables defined in the ‘.env.production‘ file and will point to `https://dem-aws-api.signicat.net` for requests against the API.

8.2.6 Environments

Different environment profiles is needed because we have several environments that the application needs to run in. There are currently two .env files in the project.

- .env (for development locally)
- .env.production (for deployment to AWS environment)

.env is the default one, and is used for development. It will automatically be used when running:
npm start

.env.production is chosen automatically when running
npm run build

8.2.7 Git-Hooks

Git hooks are used to run scripts before running git commands like push, commit and so forth. We are using a dependency called ”Husky” for having git hooks initialized for all developers in the project.

All hooks are installed and ”.husky” folder is set as default hook folder automatically when running **npm install**.

8.2.7.1 Pre-commit

Currently, Demo-application only utilizes the pre-commit hook using Husky. This hook will run unit tests before every commit.

Hooks can be found in the folder `/.husky` from the root folder.

Skipping the pre-commit hook can be done through adding argument to git commit: ‘--no-verify‘
Example: ”git commit -a -m ’commit message’ --no-verify”

8.2.8 Linter and style formatting

8.2.8.1 Eslint

Demo-application project uses Eslint for linting and has a project specific rule-set defined in a ‘.eslint.json‘ file. The linter starts up during runtime and will display linting errors in the console. Style formatting is disabled for Eslint as the project uses Prettier instead for this, and uses Eslint only as a linter.

Recommendation:
Debugging and development goes a lot faster if you are using VS Code extension for Eslint.

8.2.8.2 Prettier

Style formatting is disabled for eslint, and is done through Prettier instead. Ruleset for formatting is set in the ‘.prettierrc.json‘ file.

Recommendation:
Install Prettier VS Code extension to format on save in accordance to ‘.prettierrc.json‘.

8.2.9 Dockerfile

For the demo-application to be dockerized, we have set up a Dockerfile that works in accordance with scripts defined in package.json file.

Dockerfile uses the following multistage building pattern:

1. First stage installs dependencies and then runs the build script.
2. Second stage copies build folder into docker container and serves it using Nginx.

9 Continuous integration and testing

9.1 Backend

In the Digital Evidence Management API, there is set up a gitlab-ci.yml file which defines what stages to run, and what should be done as part of Continuous Integration on Gitlab CI.

Continuous integration is set up to use four stages:

- Build
- Test
- Deploy-containers (only on master)
- Publish (only on master)

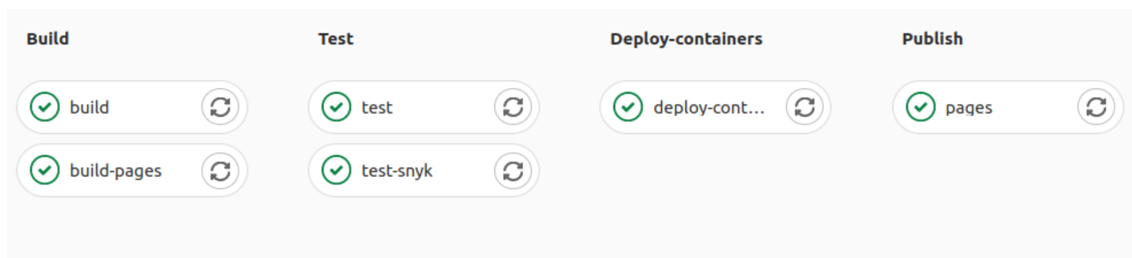


Figure 42: *Continuous Integration stages*

Build
Build stage has two parallel jobs: ”build” and ”build-pages”. In this stage ”build” job CI compiles the maven project and saves the artifacts and cache, while build-pages builds API-documentation and stores the artifacts for later use.

Test
The API contain both unit tests and integration tests. The unit tests are run in Maven by using the Maven Surefire plugin, while the integration tests are run with the Maven Failsafe plugin. Because integration tests are much slower to execute we have excluded these tests from the normal build lifecycle. The two types of tests are differentiated by the naming of the class: ”**UnitTest.java” for unit tests and ”**IT.java” for integration tests.

Test stages also has two parallel jobs: ”test” and ”test-snyk”. The job ”test” runs all integration tests and unit tests, while the job ”test-snyk” checks the project for vulnerabilities.

Deploy-containers

Deploy stage only has the one job: "deploy-containers". In that job the Gitlab CI runner builds a docker image based on the Dockerfile in the project using the artifacts stored in the "build" stage, and then pushes the image with the tag "latest" to the project's container registry. This stage is only run when there is a merge to master.

```
1 FROM amazoncorretto:15
2 ARG JAR_FILE=./web/target/*.jar
3 COPY ${JAR_FILE} app.jar
4 EXPOSE 9002
5 ENTRYPOINT ["java", "--enable-preview","-jar","/app.jar"]
```

Figure 43: *The Dockerfile used for the deployment stage.*

We also wanted to have this stage automatically restart Nomad jobs which is running on AWS, so that they would pull the latest image that we pushed to the container registry, but sadly there were complications with Gitlab CI being blacklisted by the AWS services that Signicat uses. We looked into web-hooks, and simple curl commands to trigger Jenkins jobs that were white-listed by the AWS services so that we would have Continuous Deployment as well, but sadly there was not enough time to finish this stage. So for now to really deploy to AWS, we would have to wait until the Deploy stage is complete, and then manually trigger a build for Jenkins.

```
#This is not used as Gitlab CI is not whitelisted from AWS services.
.deploy-aws:
  image: openjdk:15.0.2-jdk-buster
  stage: deploy-aws-jobs
  script:
    - apt update && apt install -y curl
    - apt-get install software-properties-common -y
    - curl -fsSL https://apt.releases.hashicorp.com/gpg | apt-key add -
    - apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb_release -cs) main"
    - apt-get update && apt-get install nomad
    - nomad job run -address="http://nomad-demo.signicat.net:4646/" dem.hcl
  only:
    - master
```

Figure 44: *The deploy stage we initially wrote for Continous Deployment*

In this stage nomad would look for a gitlab secret variable and then feed that in when running the "Nomad job run" command. This was never used as there was some red tape in getting Gitlab CI white-listed from AWS services.

Publish

Publish stage has only the one job: "pages". If all other stages passes and a new container image is pushed to the container registry in the previous stage, then the publish stage publishes new API

documentation to Gitlab Pages. This stage is only run when there is a merge to master.

9.2 Frontend

In Digital Evidence Management Demo-application repository, there is a gitlab-ci.yml file that specifies how Gitlab CI should handle the stages of building, running tests, checking project with snyk for vulnerabilities, and deploying a dockerized image to the repository's container registry on Gitlab.

Stages defined in the CI file for the Demo Application is:

- Build
- Test
- Deploy (only on master)

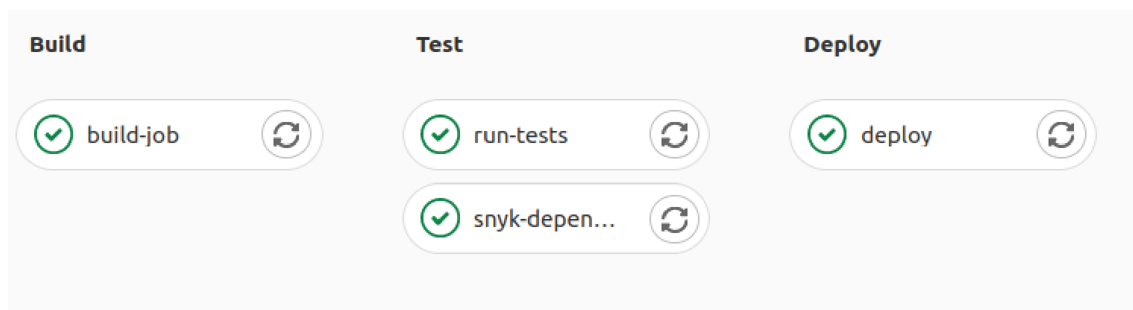


Figure 45: *Continuous Integration stages for Demo-application*

Build
Build stage installs all dependencies needed for running tests and stores the folder ”node_modules” with the dependencies in cache.

Test
Test stage retrieves ”node_modules” from cache and then runs the two jobs: ”run-tests” and ”snyk-dependency” in parallel.

The job ”run-tests” runs all unit tests and checks that every ui element is rendered and displayed to user correctly bases on user interaction. The job ”snyk-dependency” checks the project for any vulnerabilities with snyk.

Deploy
In the deploy stage the Gitlab CI runner builds a new docker image based on the Dockerfile in the

project, and then pushes it to the repository's container registry. Within the Dockerfile there are instructions for how to build a production ready deployment and how the docker container should be started. This stage is only run on a merge to master.

```
Dockerfile > ...
1  # Step 1
2  FROM node:13.12.0-alpine as build
3  RUN mkdir /app
4  WORKDIR /app
5  COPY package.json /app
6  RUN npm install --silent
7  COPY . /app
8  RUN npm run build
9
10
11 # Step 2
12
13 FROM nginx:1.17.1-alpine
14 COPY nginx.conf /etc/nginx/nginx.conf
15 COPY --from=build /app/build /usr/share/nginx/html
16 EXPOSE 80
17
```

Figure 46: *The Dockerfile used for the deployment stage.*

9.3 Testing

On every push and merge to Gitlab repository we would have two parallel jobs running for tests. One of the jobs is to run the tests written to test integration and unit tests. The other job is for checking the project dependencies for vulnerabilities. For any stage after to be able to deploy containers and publish pages, tests will need to pass. We also had the policy on not merging pipelines that had failures, as that would be an error that would ripple throughout the project.

9.3.1 API - Unit and Integration Tests

The API contain both unit tests and integration tests. We have taken care to keep unit tests as focused on the given test class, to achieve this we used the Mockito testing framework to mock any outside logic.

Integration tests are used to ensure that each component of the application is working in unity to receive the desired result. For the integration tests no mocking was done so that we could test the entire system from controller to database. To do this we used the Library RestAssured to handle our interaction with the endpoints and check against expected results. These tests are run against a separate database used only for this purpose.

We use JaCoCo to generate reports on the coverage of the tests. The module "coverage" in the project exists entirely for this purpose, as it is the only way for JaCoCo to aggregate results from multiple Maven modules.

coverage

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
web		87%		80%	22	87	41	316	10	57	2	10
core		76%		50%	34	85	14	115	19	70	3	17
mongoDB		88%		70%	15	43	12	106	5	26	0	2
Total	381 of 2,554	85%	37 of 124	70%	71	215	67	537	34	153	5	29

Figure 47: *JaCoCo test coverage of the API.*

9.3.2 Demo Application - Component Tests

Jest is a part of the react-testing-library, and we use it for automated testing of components. The flow of the tests are very similar to how an user would like to interact with the website. For example, if a user click on a table entry we would expect a modal opens up with the entry's details. We wrote the tests in this manner to check for and assert that there are no breaking changes to the normal user interaction flow when interacting with the website.

The testing is very similar to other frontend testing libraries like Enzyme and Cypress, but with Jest, tests are run without displaying the UI.

A coverage report is automatically generated when running tests, and can be found within the folder ”coverage” in the project’s repository.

All files

95.87% Statements 534/557 85.59% Branches 196/229 89.6% Functions 181/202 96.13% Lines 522/543

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements		Branches		Functions		Lines	
src		100%	37/37	100%	14/14	100%	9/9	100%	37/37
src/components/animations		100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/components/buttons		100%	16/16	100%	8/8	100%	8/8	100%	16/16
src/components/containers		100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/components/dropdown		90.91%	10/11	75%	3/4	100%	5/5	90.91%	10/11
src/components/keyvaluepairs		92.31%	24/26	91.18%	31/34	75%	6/8	92.31%	24/26
src/components/modal		97.62%	41/42	77.27%	17/22	95.45%	21/22	97.62%	41/42
src/components/navbar		100%	14/14	100%	4/4	100%	5/5	100%	14/14
src/components/placeholder		100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/components/search		100%	39/39	96.67%	29/30	92.86%	13/14	100%	38/38
src/components/table		90.91%	40/44	76.47%	13/17	95%	19/20	90.91%	40/44
src/components/text		100%	16/16	100%	0/0	100%	8/8	100%	14/14
src/contexts		100%	4/4	100%	0/0	11.11%	1/9	100%	4/4
src/pages/ComponentGallery		100%	10/10	100%	0/0	100%	5/5	100%	10/10
src/pages/ErrorPages		50%	1/2	100%	0/0	0%	0/1	50%	1/2
src/pages/Login		85.71%	36/42	56.25%	9/16	85.71%	6/7	85.71%	36/42
src/pages/MainPage		98.61%	71/72	88.89%	16/18	100%	16/16	98.59%	70/71
src/pages/ModalPages		97.96%	48/49	75%	15/20	94.74%	18/19	97.87%	46/47
src/pages/ModalPages/recordModalComponents		94.85%	92/97	88.1%	37/42	88.57%	31/35	96.67%	87/90
src/pages/TestDataGenerator		96.3%	26/27	100%	0/0	87.5%	7/8	96.15%	25/26
src/tests/mockdata		100%	3/3	100%	0/0	100%	0/0	100%	3/3

Code coverage generated by istanbul at Wed May 12 2021 09:46:18 GMT+0200 (Central European Summer Time)

Figure 48: *Coverage report generated when running npm test*

9.3.3 Snyk

Snyk is a automated tool that checks Snyk’s database for any reported vulnerability within the project, and reports back with which version of a dependency the vulnerability lies in, and versions that patches the vulnerability. If Snyk finds any vulnerabilities then the test will return a test failure to the pipeline.

References

[1] OWASP.org. *OWASP top ten standard awareness document*. URL: `https://owasp.org/www-project-top-ten/`. retrieved 12.05.21.

8.2 Benchmarking Results

Benchmark results

- The benchmarking environment
  - Which endpoints do we want?
  - System specs.
- Load situations:
- What information we want to record:

API TESTS
- Springboot. 5min. 100 users constantly.
- Quarkus. 5min. 100 users constantly.
- Springboot. 5min. 200 users constantly.
- Quarkus. 5min. 200 users constantly.

DATABASE TESTS
- MySQL. Springboot. 5 min. 100 concurrent users.
- MySQL. Quarkus. 5 min. 100 concurrent users.
- Postgres. Springboot. 5 min. 100 concurrent users.
- Postgres. Quarkus. 5 min. 100 concurrent users.
- MongoDB. Springboot. 5 min. 100 concurrent users.
- QUERY Comparison. Springboot. 1 Concurrent user.
- Postgres without Query. Quarkus. 5 min. 100 concurrent users.
- Postgres with Query. Quarkus. 5 min (canceled after ~18 min). 100 concurrent users.
- MySQL without Query. Quarkus. 5 min. 100 concurrent users.

The benchmarking environment

Which endpoints do we want?

Method	description	endpoint
GET	Get single record based on UUID	/records/{uuid}
POST	Create new record based on body	/records
POST	Search for record with query body	/records/query
DELETE	Mark for deletion based on UUID	/records/{uuid}

System specs.

Component	Specs
Processor	Intel(R) Core(TM) i7-865U CPU @ 1.80GHz
Memory	16GB
Type	Notebook

Load situations:

Load situation	Number of users	Step count	Rampup time
Low load	100	1	5m

What information we want to record:

- Requests handled per minute
- Average request time
- Dropped requests

- Query response time
- Method response time
- Boot time

Number of records:

1 000 000

Springboot. 100 Requests per second.

Request	Time	Throughput	Error %	Min	Max	Average
GET	3m	100.5/sec	0.00%	0	13	1
POST	3m	100.2/sec	0.00%	3	18	1
QUERY	3m	100.5/sec	0.00%	0	664	6
DELETE	3m	100.4/sec	0.00%	0	45	2

API TESTS

Springboot. 5min. 100 users constantly.

SPRING BOOT	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	1083735	7	7	13	16	26	0	496	0.0	3604.1/sec
Post Request	1083708	6	5	11	14	24	0	371	0.0	3610.2/sec
Query Request	1083679	5	5	11	14	24	0	370	0.0	3611.9/sec
Delete Request	1083659	5	5	10	13	23	0	369	0.0	3612.6/sec
TOTAL	**4334781**	**6**	**6**	**11**	**14**	**25**	**0**	**496**	**0.0**	**14415.9/sec**

Quarkus. 5min. 100 users constantly.

QUARKUS	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	560817	50	39	105	131	187	0	540	0.0	1868.1/sec
Post Request	560721	1	1	3	4	9	0	69	0.0	1868.1/sec
Query Request	560719	1	1	3	4	10	0	74	0.0	1868.1/sec
Delete Request	560718	0	0	1	2	7	0	46	0.0	1868.2/sec
TOTAL	**2242975**	**13**	**1**	**48**	**77**	**139**	**0**	**540**	**0.0**	**7471.5/sec**

Springboot. 5min. 200 users constantly.

SPRING BOOT	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	1145186	21	12	23	31	72	0	1322	0.0	3808.2/sec
Post Request	1145135	10	9	19	26	44	0	425	0.0	3808.4/sec
Query Request	1145075	10	8	18	25	44	0	427	0.0	3808.2/sec
Delete Request	1145020	9	8	18	23	42	0	424	0.0	3808.1/sec
TOTAL	**4580416**	**12**	**9**	**20**	**26**	**47**	**0**	**1322**	**0.0**	**15231.8/sec**

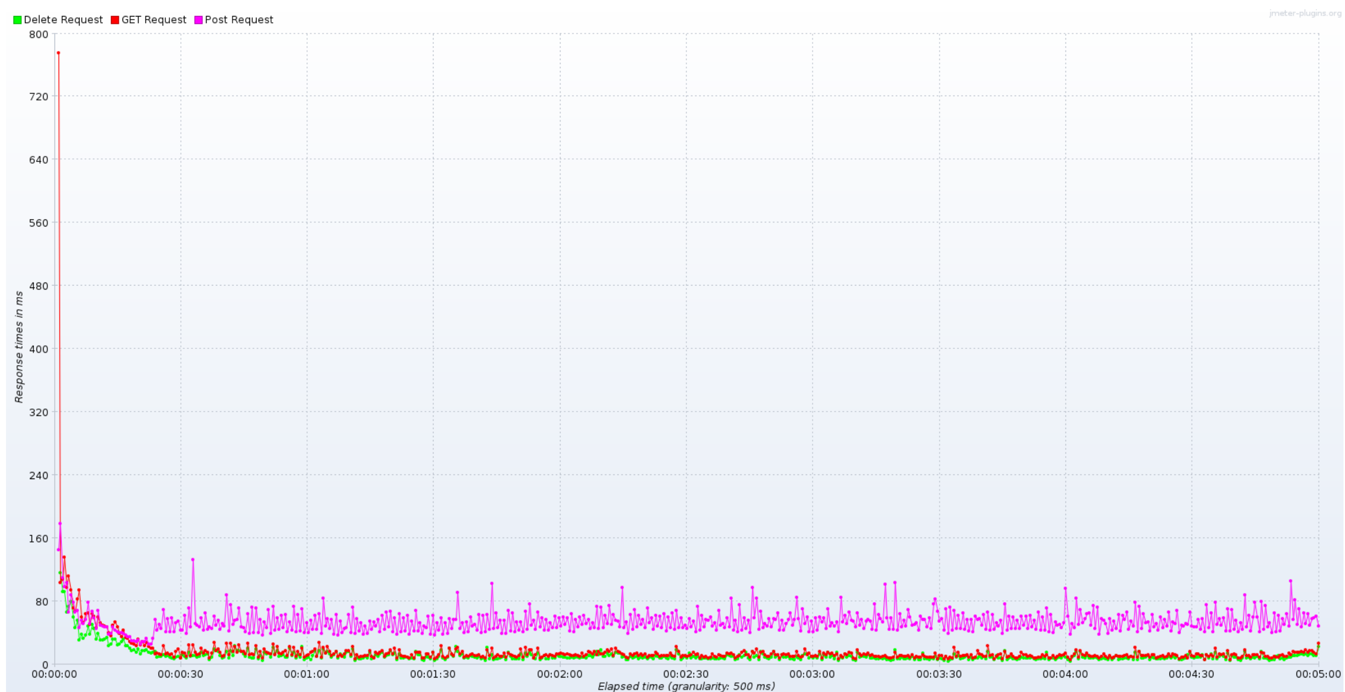
Quarkus. 5min. 200 users constantly.

QUARKUS	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	1186905	27	11	25	40	360	0	80030	0.0	1850.8/sec
Post Request	1186860	15	8	19	29	294	0	80031	0.0	1850.8/sec
Query Request	1186798	14	7	19	28	292	0	80031	0.0	1850.8/sec
Delete Request	1186746	13	7	18	26	290	0	80031	0.0	1850.8/sec
TOTAL	**4747309**	**17**	**8**	**21**	**31**	**302**	**0**	**80031**	**0.0**	**7402.8/sec**

DATABASE TESTS

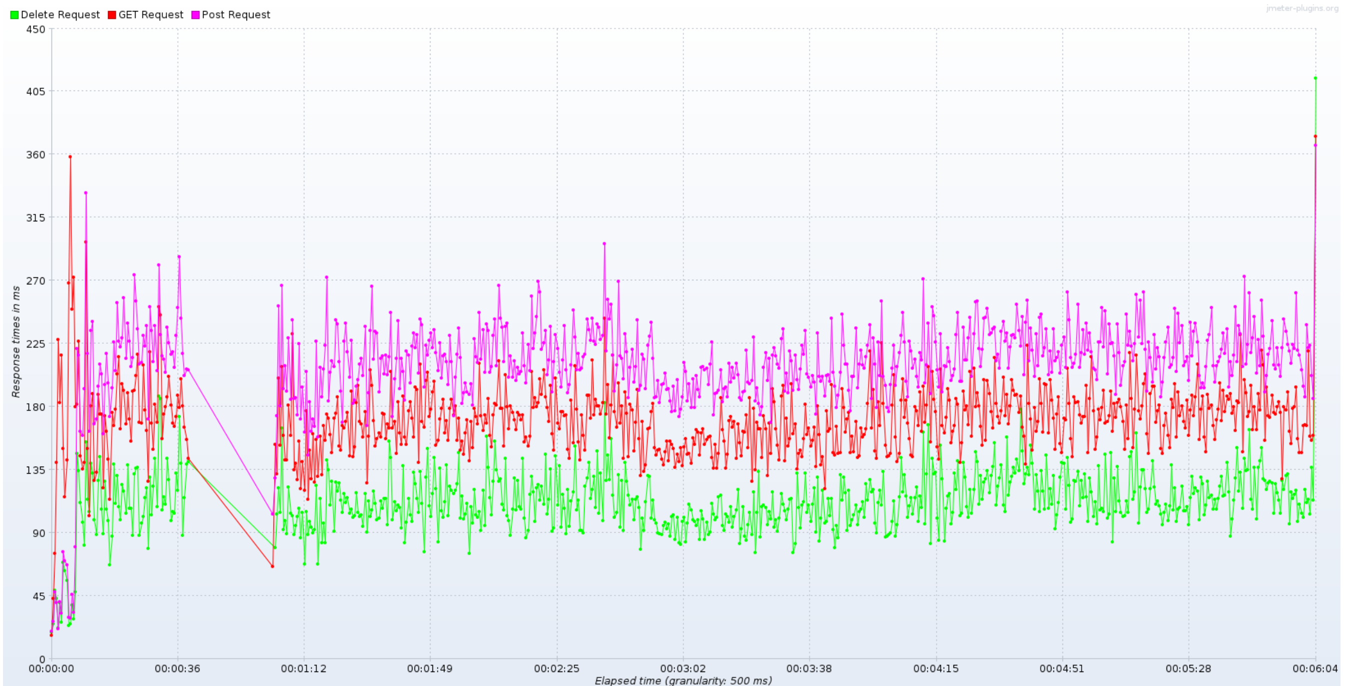
MySQL. Springboot. 5 min. 100 concurrent users.

MYSQL	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	393830	13	10	26	37	67	0	823	0.0	1312.7/sec
Post Request	393803	51	45	78	98	151	4	613	0.0	1315.9/sec
Delete Request	393754	10	6	22	31	56	0	506	0.0	1316.5/sec
TOTAL	**1181387**	**25**	**14**	**55**	**69**	**118**	**0**	**823**	**0.0**	**3937.5/sec**



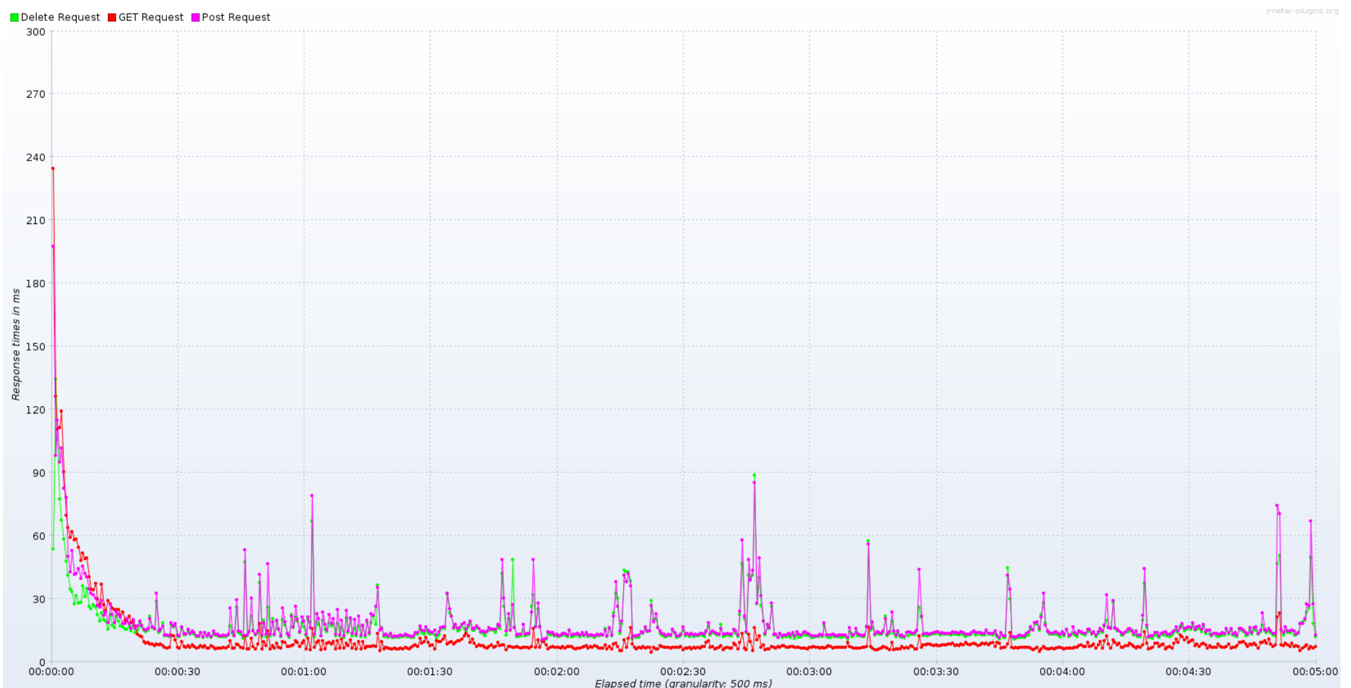
MySQL. Quarkus. 5 min. 100 concurrent users.

MySQL	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	69754	168	199	273	310	452	2	881	0.0	191.3495749402667
Post Request	69673	206	235	307	343	492	6	1093	0.01	191.1955588119877
Delete Request	69606	109	55	270	298	421	1	924	0.00	191.01379516634057
TOTAL	209033	161	193	289	320	463	1	1093	0.01	573.4173846968363



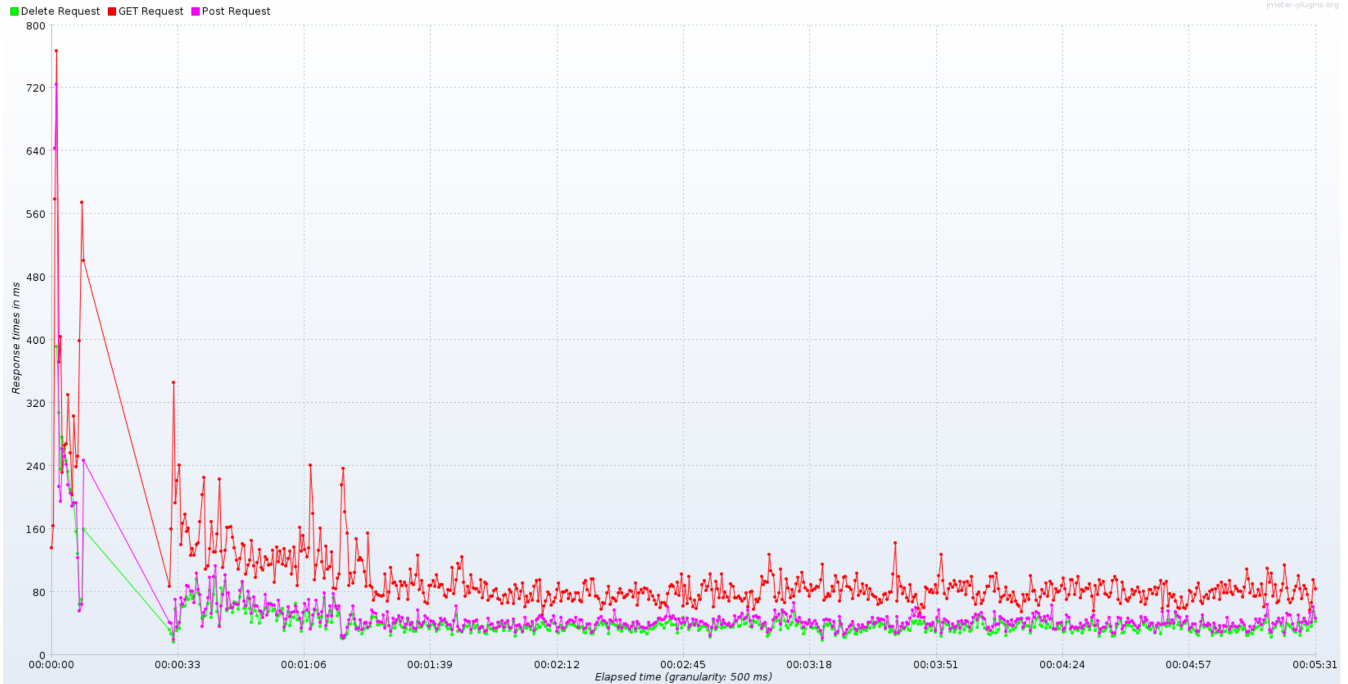
Postgres. Springboot. 5 min. 100 concurrent users.

Postgres	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	2029139	10	8	23	35	56	0	633	0.0	33.2/sec
Post Request	2028866	36	34	65	85	136	1	822	0.0	33.2/sec
Delete Request	2028794	11	9	22	32	56	0	613	0.0	33.2/sec
TOTAL	**6086799**	**19**	**12**	**46**	**58**	**104**	**0**	**822**	**0.0**	**99.5/sec**



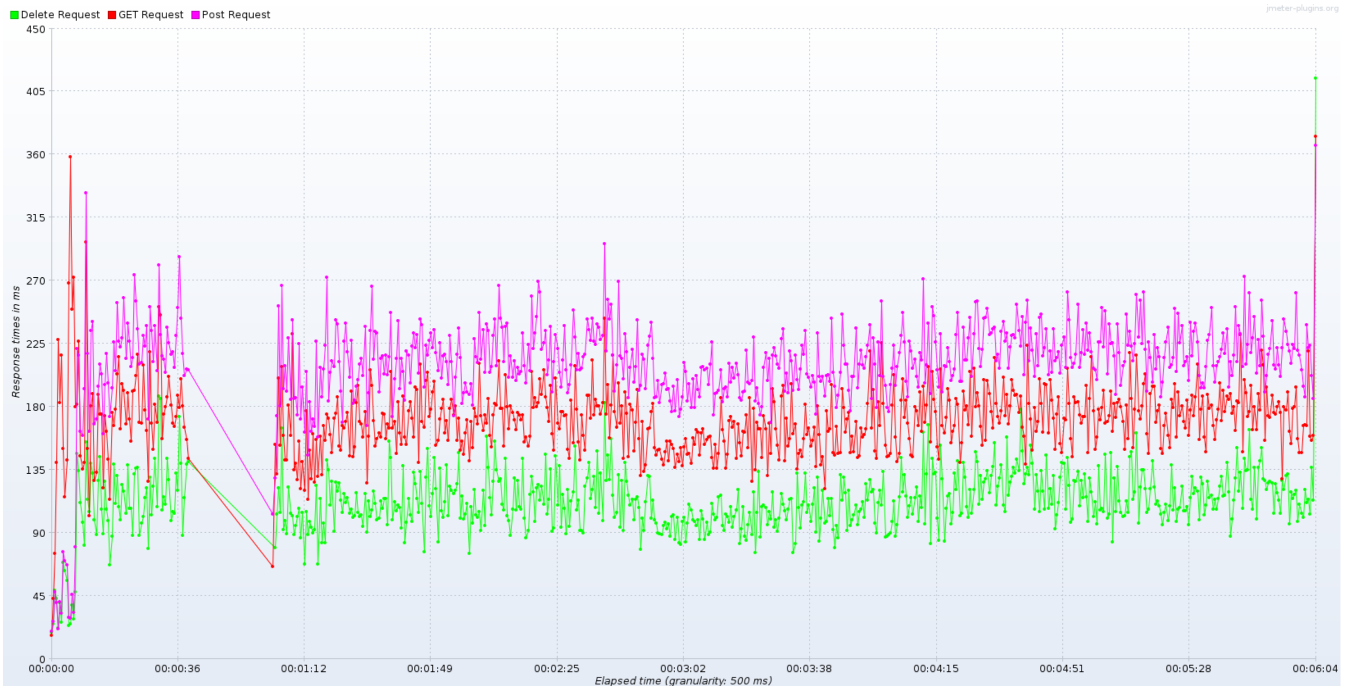
Postgres. Quarkus. 5 min. 100 concurrent users.

MongoDB	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	360659	87	76	165	206	305	2	1047	0.1	1090.3/sec
Post Request	360458	43	28	94	120	190	4	1230	0.0	1090.3/sec
Delete Request	360421	37	23	85	109	174	1	728	0.0	1092.8/sec
TOTAL	1081538	56	39	123	157	251	1	1230	0.0	3269.5/sec



MongoDB. Springboot. 5 min. 100 concurrent users.

MongoDB	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	1208366	19	19	30	35	49	1	307	0.0	4027.3/sec
Post Request	1208274	14	15	24	28	41	0	398	0.0	4027.9/sec
Delete Request	1208222	14	14	25	30	49	0	293	0.0	4029.2/sec
TOTAL	**3624862**	**16**	**16**	**27**	**32**	**47**	**0**	**398**	**0.0**	**12080.9/sec**

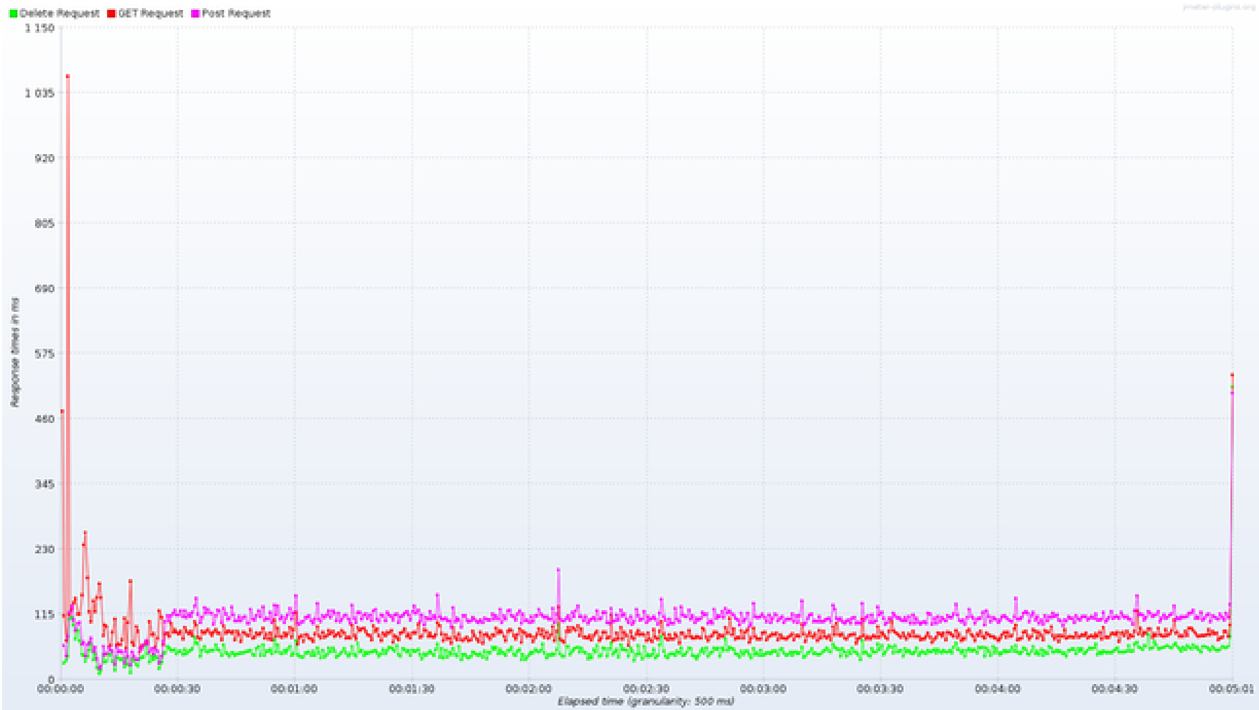


QUERY Comparison. Springboot. 1 Concurrent user.

QUERY REQUESTS	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Throughput
MySQL - Spring									
Round 1	516	1192	1193	1275	1318	1481	615	1675	1.6/sec
Round 2	504	1192	1190	1262	1313	1428	1070	1791	1.7/sec
Average	**510**	**1192**	**1191,5**	**1268,5**	**1315,5**	**1454,5**	**842,5**	**1733**	**1.65/sec**
MySQL - Quarkus									
Round 1	414	1605	1609	1737	1759	1862	91	1876	1.1/sec
Round 2	382	1573	1540	1702	1765	2163	1222	2429	1.3/sec
Average	**398**	**1589**	**1574,5**	**1719,5**	**1762**	**2012,5**	**656,5**	**2152,5**	**1.2/sec**
Postgres - Spring									
Round 1	466	1301	1292	1422	1436	1484	44	1583	1.5/sec
Round 2	446	1347	1341	1494	1506	1611	1137	1653	1.5/sec
Average	**456**	**1324**	**1316,5**	**1458**	**1471**	**1547,5**	**590,5**	**1618**	**1.5/sec**
Postgres - Quarkus									
Round 1	958	1350	1338	1468	1505	1656	657	2041	0.6/sec
Round 2	215	1397	1383	1506	1523	1601	1172	1705	0.7/sec
Round 3	625	1442	1410	1592	1711	2144	1172	2205	0.8/sec
Average	**599**	**1396**	**1377**	**1522**	**1580**	**1800**	**1000**	**1984**	**0.7/sec**
MongoDB - Spring									
Round 1	756	793	788	850	862	927	694	1311	2.5/sec
Round 2	754	795	793	842	873	1008	689	1292	2.5/sec
Average	**755**	**794**	**790,5**	**846**	**867,5**	**967,5**	**691,5**	**1301,5**	**2.5/sec**
MongoDB - Quarkus									
Round 1	391	768	743	833	889	1103	664	2986	1.3/sec
Round 2	392	765	735	854	931	1083	665	1286	1.3/sec
Average	**391,5**	**766,5**	**739**	**843,5**	**910**	**1093**	**664,5**	**2136**	**1.3/sec**

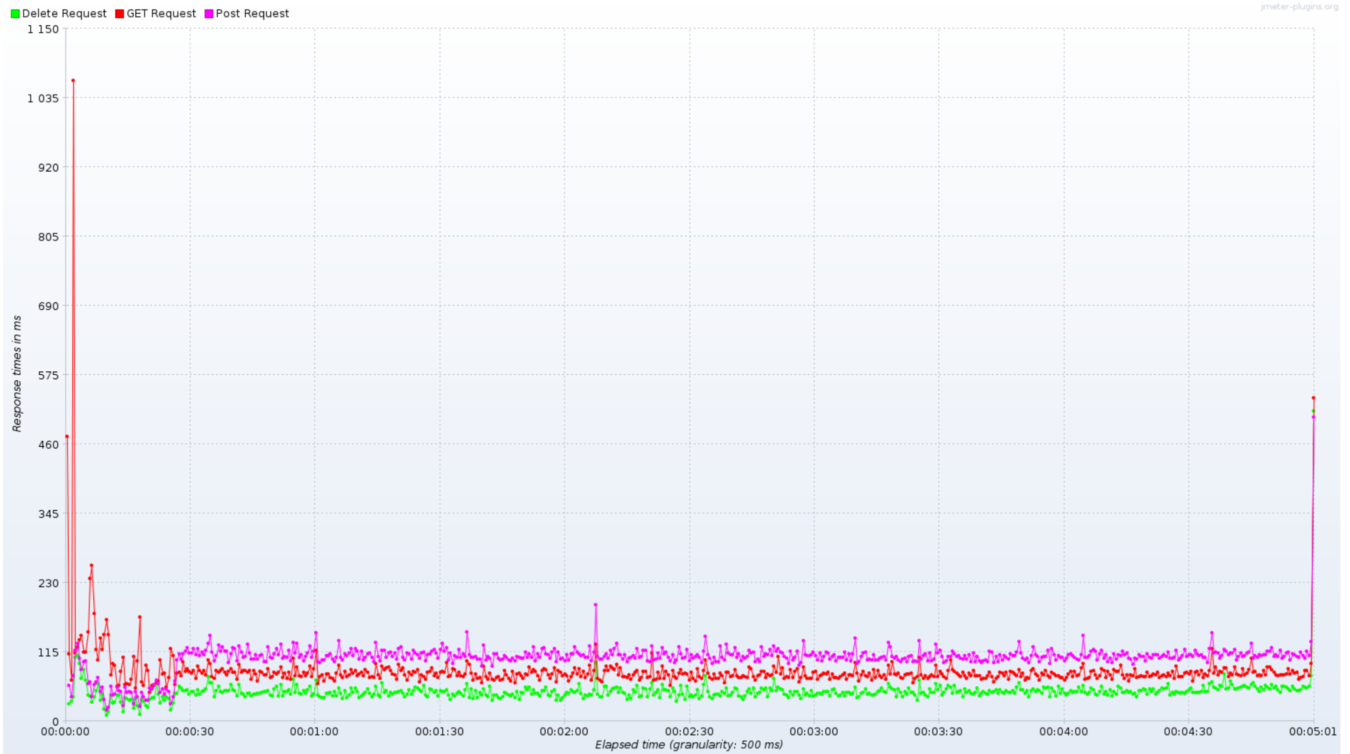
Postgres without Query. Quarkus. 5 min. 100 concurrent users.

Postgres	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	270415	56	47	115	145	221	1	833	0.0	900.8/sec
Post Request	270377	31	17	75	95	145	3	575	0.0	902.5/sec
Delete Request	270339	22	9	63	82	128	1	346	0.0	903.2/sec
TOTAL	**811131**	**36**	**21**	**88**	**113**	**179**	**1**	**833**	**0.0**	**2702.1/sec**



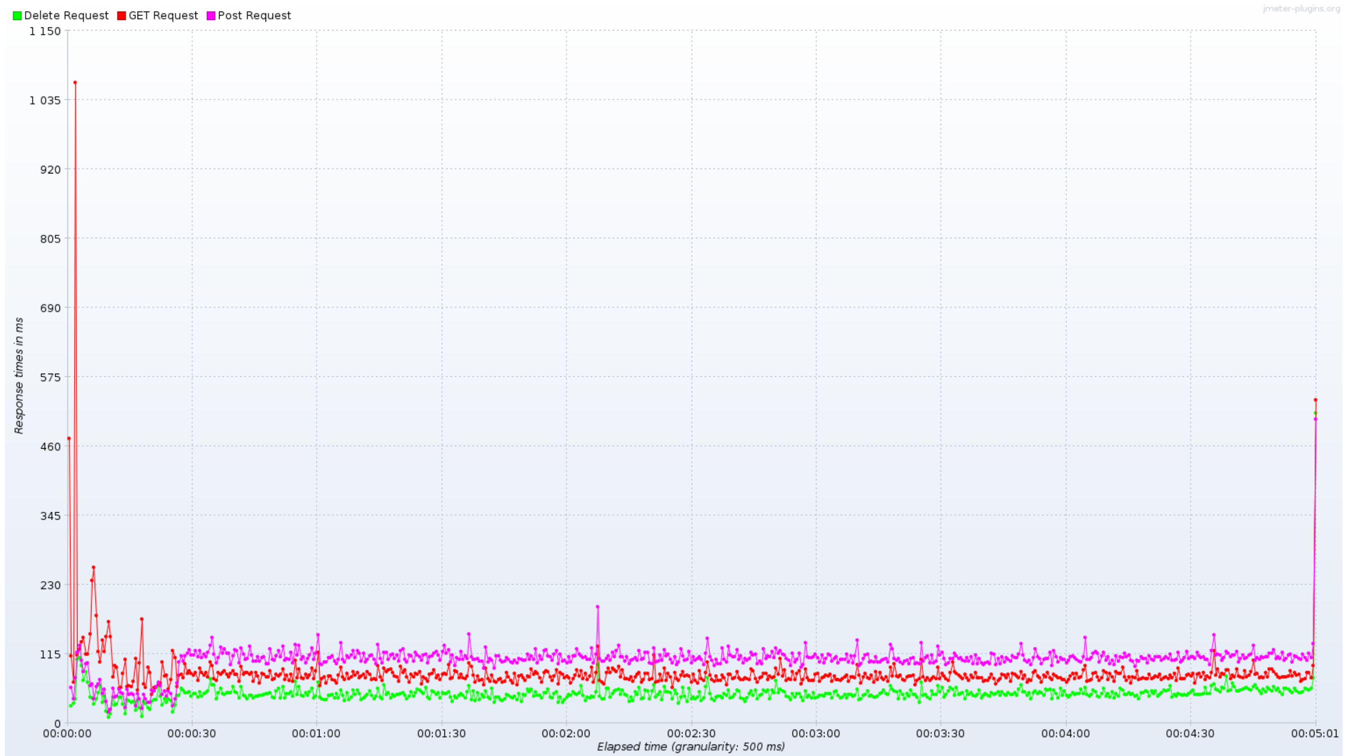
Postgres with Query. Quarkus. 5 min (canceled after ~18 min). 100 concurrent users.

Postgres	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	216	11812	169	14874	23525	354804	9	526696	21.29	16.2/min
Post Request	192	50615	8015	21616	453012	721090	5	786665	54.17	14.4/min
Query Request	100	204268	14640	622375	720817	1061771	6456	1061771	100	5.6/min
Delete Request	52	274557	20898	774893	1062844	1063350	212	1063350	88.46	2.9/min
TOTAL	**560**	**83881**	**6869**	**360402**	**621422**	**798309**	**5**	**1063350**	**52.86**	**31.3/min**



MySQL without Query. Quarkus. 5 min. 100 concurrent users.

Postgres	Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
GET Request	259646	80	85	140	173	240	1	1988	0.0	863.1/sec
Post Request	259584	102	112	165	194	253	6	611	0.0	864.5/sec
Delete Request	259480	47	15	114	137	202	1	573	0.0	864.7/sec
TOTAL	**778710**	**76**	**82**	**146**	**174**	**237**	**1**	**1988**	**0.0**	**2588.7/sec**


Delete Request
GET Request
Post Request
jmeter-plugins.org
Response times in ms
1 150
1 035
920
805
690
575
460
345
230
115
0
00:00:00
00:00:30
00:01:00
00:01:30
00:02:00
00:02:30
00:03:00
00:03:30
00:04:00
00:04:30
00:05:01
Elapsed time (granularity: 500 ms)

8.3 Requirement Documentation

Requirement Documentation:

Digital Evidence Management

Joakim Moe Adolfsen William Jarbeaux Thomas Bakken Moe Eric Younger

Spring 2021

TDAT 3001 - Group 109
version 1.0

Audit history

Date	Version	Description	Author
13.01.2021	0.1	First draft	Joakim Moe Adolfsen, William Jarbeaux, Thomas Bakken Moe, Eric Younger
11.05.2021	1.0	Added requirements and wireframes	Joakim Moe Adolfsen

Contents

1 Introduction **4**

2 Requirements **5**

2.1 Functional Requirements 6

2.2 Non-functional Requirements API 7

2.2.1 Infrastructure and dependencies 7

2.2.2 Application 8

2.2.3 CI/CD 9

2.2.4 Security 9

2.3 Non-functional Requirements - Demo Application 10

2.3.1 Infrastructure and dependencies 10

2.3.2 Application 10

2.3.3 CI/CD 10

2.3.4 Security 11

3 User stories **12**

4 Prototypes **13**

4.1 Wireframes 13

1 Introduction

This document is written for the bachelor project in the computer engineering programme (ITHINGDA) at the Norwegian University of Science and Technology. Our bachelor thesis concerns the development of a system called ”Digital Evidence Management” on behalf of our client Signicat. This document serves as an attachment to our main thesis.

The main function of this document is to layout the requirements of the assignment. In our case this will be the intended behaviour of the system we are developing.

2 Requirements

This section will document the requirements given to us by Signicat at the beginning of the project. These requirements consist of a set of functional requirements for the project and then non-functional requirements for the API and the Demo Application.

It is worth noting here that these requirements were given before the scope of the assignment was changed from a fully developed system to more of a test environment.

2.1 Functional Requirements

Requirement	Description	Commment
Storage time	Up to 20 years	
Number of records	500 millions	In one system
REST API	Create record, Search records, Get record, Validate record, Delete record (mark for delete)	All records will have unique ID (GUID)
Flexible metadata	Customer can define own metadata	These should be searchable and not case sensitive at least for attribute bame
Admin API	Create customer, Delete all customer data, Retrieve all customer data, Number of records per customer	
System metadata	Time and date from TSA, Delete grace period, Mark for delete (time/date or empty), Valid until, Validity of record related to timestamp (time/date), Reference	
Physical delete	Type cron job deleting	Only for Signicat
REST Search function	Powerful query to be constructed by customer. Only search in metadata including system metadata. Must be fast (indexed metadata)	No validation
Demo application	Including source code	Github?
Preservation	Preservation cron job A record will be valid for about 3 years according to timestamp	Could be same as delete cron job. Only Signicat. Could we avoid this and make records "valid" only for 3 first years?
Reference between records	When creating a record it should be possible to refer another record as GUID.	Should one record refer multiple records or just one?

2.2 Non-functional Requirements API

2.2.1 Infrastructure and dependencies

Non-Functional requirements	**Description**	**Comment**
Storage	MUST use MySQL	API needs search, we need consistency (ACID)
Deployment platform	MUST target Kubernetes	Deployment files are the team's responsibility - not the k8s cluster IaC itself
Billing	MUST support Signicat Billing	evaluate micro-billing

2.2.2 Application

Non-Functional requirements	Description	Comment
Programming language	MUST be Java (latest stable with preview enabled, which will close some of the gap between Kotlin and older Java)	PoC implemented in Kotlin. Adding Kotlin to Signicat Green Stack's set of languages was discussed in a technology alignment meeting. (Languages are per stack, not per team or project.)
Packaging	MUST release containers	Build as a Spring boot or Quarkus application.
Versioning	MUST support versioning for continuous deployment	Defined in Versioning for Continuous deployment Deployment at Signicat
HTTP API	MUST be REST	
Signicat rest API guidelines	MUST implement the Signicat rest API guidelines	https://signicat.gitlab.io/architecture-group/rest-api-guidelines/introduction
Unit test code coverage	MUST be above 90 percent	
Linting	MUST respect linting	TODO, Options: point to a checkstyle.xml point to a sonarcloud config Which to rely upon?
Sonarcloud	MUST publish to Sonarcloud at least on each Sonarcloud merge to master	Integrate with Sonarcloud
Snyk	MUST have no vulnerabilities	Integrate with snyk.io
API Documentation	MUST offer a swagger/openapi definition	Make sure that documentation is good as https://beta.developer.signicat .com/ uses swagger
Metrics	MUST expose metrics	See existing Java apps for relevant standard metrics
Logs	MUST log to stdout MUST integrate with centralized logging MUST support deleting all logs for a given customer	

2.2.3 CI/CD

Non-Functional requirements	Description	Comment
Build and deployment	MUST be built and deployed with gitlabci	
Merge to master	MUST be merged to master by employees	Take advantage of gitlab's support for only allowing Maintainers to merge Developer's PR Consider a long lived developer branch that the students use as "their" master.

2.2.4 Security

Non-Functional requirement	Description	Comment
GDPR compliance	MUST be compliant with GDPR	Among other things, do not log things that contain PII.
OIDC	MUST secure endpoints with OIDC	TODO, do we require OIDC on all endpoints
Tenant isolation	MUST design system to guarantee good tenant isolation MUST be safe to remove tenants from live systems	Several isolation model exist at deployment level: 1 set of service 1 database. That's what signicat green traditionally does. in this case software is responsible of isolation 1 set of service n database. Giving one database per customer would be great in terms of isolation (when customer leaves we drop the whole db) M set of services N database.
Encryption in transit	MUST use TLS in exposed APIs	
Encryption at rest	MUST make use of encryption at storage backend	Multi tenancy design can lead to discussions relative to one set of encryption keys per database Key management is hard Explore support in managed services

2.3 Non-functional Requirements - Demo Application

2.3.1 Infrastructure and dependencies

Non-Functional requirements	Description	Comment
Deployment platform	MUST target Kubernetes	Deployment files are the team's responsibility - not the k8s cluster IaC itself

2.3.2 Application

Non-Functional requirements	Description	Comment
Programming language	MUST be Javascript/CSS/HTML	Decide specific JS framework.
Packaging	MUST release containers	
Versioning	MUST support versioning for continuous Defined in Versioning for Continuous deployment	Defined in Versioning for Continuous deployment Deployment at Signicat
Unit test code coverage	MUST be above 85	
Linting	MUST respect linting	ESLint
Sonarcloud	MUST publish to Sonarcloud at least on each merge to master	TODO, does it makes sense?
Snyk	MUST have no vulnerabilities	TODO, does it makes sense?

2.3.3 CI/CD

Non-Functional requirements	Description	Comment
Build and deployment	MUST be built and deployed with gitlabci	
Merge to master	MUST be merged to master by employees	Take advantage of gitlab's support for only allowing Maintainers to merge Developer's PR Consider a long lived developer branch that the students use as "their" master.

2.3.4 Security

Non-Functional requirements	Description	Comment
GDPR compliance	MUST be compliant with GDPR	
WCAG 2.1	MUST support WCAG 2.1	TODO, which standard to support?

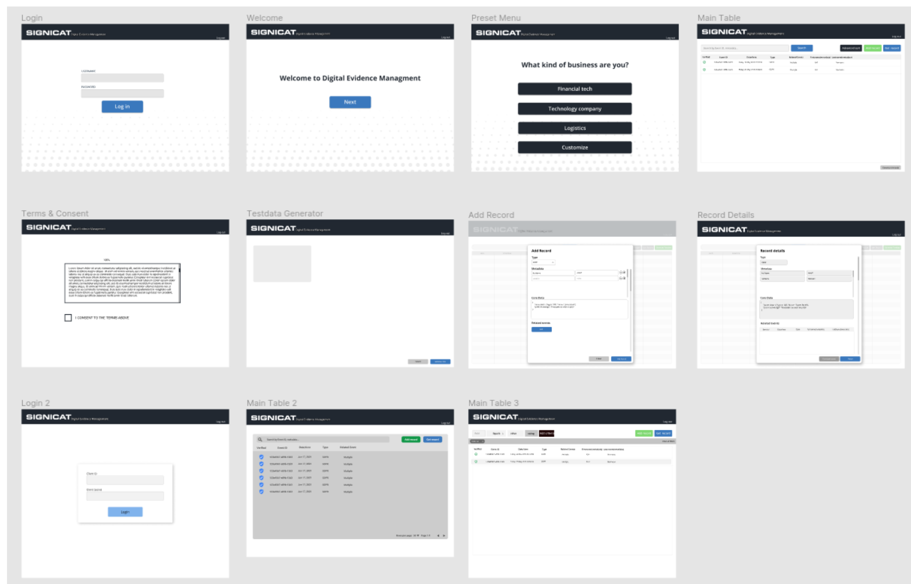
3 User stories

As a	I want to...	so that...
User	view my stored records in an organized way	I know how many records I have and what they contain.
User	search through my stored records with various conditions	I can efficiently find records that I am looking for.
User	search through my stored records with various conditions	I can efficiently find records that I am looking for.
User	be able to find relations between records	It is easier to find relevant data.
User	have my data secure(encrypted)	other people can't misuse my stored data.
User	have my data isolated	my data is secure and not viewed by others.
User	have detailed and clear API documentation	I can easily start using the API.
Stakeholder	record billable events	it is easy to send accurate invoices.
Maintainer	add authentication	the system is only accessible by our customers.
Maintainer	add centralized logging	I am able to trace errors, and store logs for future needs.
Developer	add timestamping	I am sure that my data has not been tampered.
Maintainer	add metric data collection	I can assess how the system is doing while in production.
Developer	have unit tests	refactoring, or modifications to code will be automatically tested to see if the code changes results in breaking parts of the system.

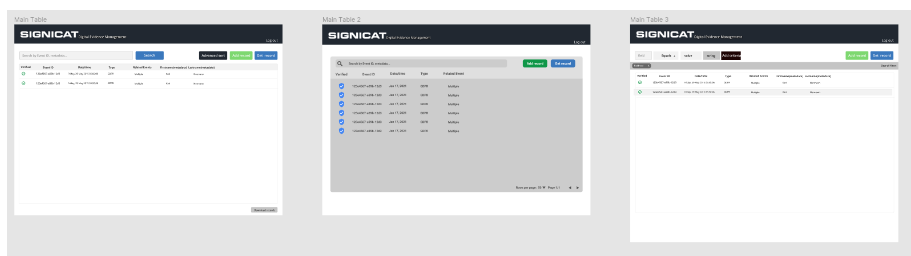
4 Prototypes

4.1 Wireframes

After receiving the initial requirement documents we created some wireframes to plan the design of the Demo Application .



All wireframe screens.



Wireframes of the main table. The first two are variations considered during development, while the third is closest to the final result after the redesign of the search module.

To view the full wireframe see:
`https://www.figma.com/file/wn648S7W2xZ9w2PbtNBgHB/Digitial-Evidence-Managment-wireframe-v1.0?node-id=0%3A1`

8.4 Vision Documentation

Vision document:
Digital Evidence Management

Joakim Moe Adolfsen William Jarbeaux Thomas Bakken Moe Eric Younger

Spring 2021

TDAT 3001 - Group 109
version 1.0

Audit history

Date	Version	Description	Author
12.01.2021	0.1	First draft	Joakim Moe Adolfsen, William Jarbeaux, Thomas Bakken Moe, Eric Younger
13.01.2021	0.2	Complete first draft	Joakim Moe Adolfsen, William Jarbeaux, Thomas Bakken Moe, Eric Younger
25.01.2021	0.3	Revisements from supervisor	Joakim Moe Adolfsen, William Jarbeaux, Thomas Bakken Moe, Eric Younger
20.05.2021	1.0	Final version	Joakim Moe Adolfsen, William Jarbeaux, Thomas Bakken Moe, Eric Younger

Contents

1 Introduction **5**

2 Problem statement **6**

2.1 Problem 6

2.2 Product 6

3 Stakeholders and users **7**

3.1 Stakeholders 7

3.2 Users 7

3.3 User environment 7

3.4 Summary of user needs 8

3.4.1 In project 8

3.5 Alternatives to our product 8

4 Overview **9**

4.1 Role in the user environment 9

4.2 Assumptions and dependencies 9

5 Functional requirements **10**

6 Non-functional requirements **11**

6.1 Back-end 11

6.1.1 Infrastructure and dependencies 11

6.1.2 Application 12

6.1.3 CI/CD 13

6.1.4 Security 13

6.2 Front-end 14

6.2.1 Infrastructure and dependencies 14

6.2.2 Application 14

6.2.3 CI/CD 14

6.2.4 Security 15

1 Introduction

The purpose of this document is to create an outline for the project. This is to help developers and stakeholders form a common understanding of the project and its scope.

The task given is to create a digital evidence management system. The core of this system is an API, but for the sake of demonstration it will also contain a front-end. This API is needed to store evidence of consent, transactions, and more for high security applications. The API should be a flexible system which allows for different types of evidence management, all depending on the companies requirements. Typical customers for this digital evidence API are financial companies, insurance companies, and other companies that handle sensitive information and consent. The demand for such an API was created by customers that wanted a way to store evidence of consent in a secure way. The evidence should also have the ability to be stored long term, and have the ability to be retrieved at a later date when the evidence is needed.

2 Problem statement

2.1 Problem

Problem:	There exists no efficient solutions to store, navigate, search, and manage digital evidence.
Involves:	Digital companies that require evidence management of anything from user GDPR consent messages to bank transactions.
Result:	Companies will have to develop in-house solutions.
Successful solution:	Offers a finished product that will be cheaper than for the company to develop their own, that will be maintained, and is compliant with all the required rules and regulations.

2.2 Product

Made for:	Digital companies.
That requires:	Digital Evidence Management.
The product:	Is a data handler system.
Goal for product:	To be cheaper and simpler way for the clients to manage their digital data.
As opposed to:	In-house solutions
Our product:	Will be a customizable standard solution that will be supported and maintained.

3 Stakeholders and users

3.1 Stakeholders

Name	**Description**	**Role during development**
Signicat	The client that defined the task. Product owner that wants to develop the application to sell it as a product. Represented by Tor Even Dahl.	Client, Product Owner, Maintainer.
NTNU	The Norwegian University for Science and Technology. The uni. where group 109 are currently enrolled. Represented by Nils Tesdal.	Supervisory role, will be grading the project based on process, product and documentation/report.
Bachelor group 109	The four students that will be developing the product and writing the bachelor thesis.	Developers of the product, and writers of the documentation

3.2 Users

Name	**Role during development**	**Represented by**
Students/Developers	Responsible for planning and development of the application.	Bachelor group 109
Product Owner	Manage workflow and task prioritization.	Elisabeth Ulsund (Signicat)
Tech Lead	Final say on technology decisions.	Steinar Knutsen (Signicat)

Name	**Role during development**	**Represented by**
Students/Developers	Responsible for planning and development of the application.	Bachelor group 109
Product Owner	Manage workflow and task prioritization.	Elisabeth Ulsund (Signicat)
Tech Lead	Final say on technology decisions.	Steinar Knutsen (Signicat)

3.3 User environment

Our system is going to a mostly stand-alone API which authenticated customers are free to make calls to. The system will however have to be integrated with Signicat's billing system. The system will also use an external hosted database service. The exact choice of which service to use is a point that will be decided during development.

3.4 Summary of user needs

3.4.1 In project

Need	Priority	Today's solution	Recommended solution
Save digital evidence as records	High	Various in-house systems.	Create a system to safely store data to be managed by the user.
Search through records	High	Various in-house systems.	A efficient way to search through and find relevant records.
Verify records	High	Various in-house systems.	A system that can verify integrity of records.

3.5 Alternatives to our product

We have found no commercial alternative to our product.
According to Tor Even Dahl, several customers of Signicat are looking for a product like ours, but are coming up short. Currently, if a customer wants a product that fulfills the same requirements as ours, they have to develop a solution in-house. This can prove costly both in terms of money and time for the customer, as software development might not be their forte.

4 Overview

4.1 Role in the user environment

There are today no other commercial solutions to digital evidence management available. The demand for such systems is rising and many companies are forced to develop their own implementations.

The system is primarily meant to be independent and to be easily implemented by various different clients.

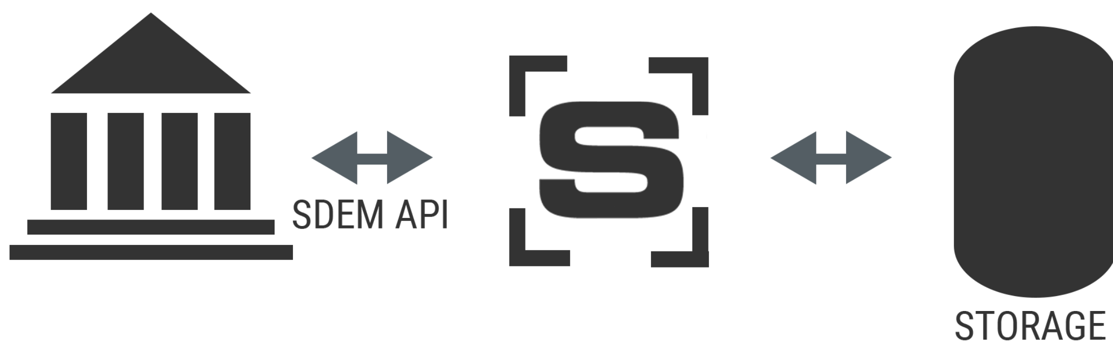


Figure 1: SDEM (Signicat Digital Evidence Management).

4.2 Assumptions and dependencies

- Signicat wants to continue the development with the properties and technologies currently presented.
- A competitor emerging during development might affect the system requirements.

5 Functional requirements

Functional requirements	**Description**	**Commment**
Storage time	Up to 20 years	
Number of records	500 millions	In one system
REST API	Create record, Search records, Get record, Validate record, Delete record (mark for delete)	All records will have unique ID (GUID)
Flexible metadata	Customer can define own metadata	These should be searchable and not case sensitive at least for attribute bame
Admin API	Create customer, Delete all customer data, Retrieve all customer data, Number of records per customer	
System metadata	Time and date from TSA, Delete grace period, Mark for delete (time/date or empty), Valid until, Validity of record related to timestamp (time/date), Reference	
Physical delete	Type cron job deleting	Only for Signicat
REST Search function	Powerful query to be constructed by customer. Only search in metadata including system metadata. Must be fast (indexed metadata)	No validation
Demo application	Including source code	Github?
Preservation	Preservation cron job A record will be valid for about 3 years according to timestamp	Could be same as delete cron job. Only Signicat. Could we avoid this and make records “valid” only for 3 first years?
Reference between records	When creating a record it should be possible to refer another record as GUID.	Should one record refer multiple records or just one?

6 Non-functional requirements

6.1 Back-end

6.1.1 Infrastructure and dependencies

Non-Functional requirements	**Description**	**Comment**
Storage	MUST use MySQL	API needs search, we need consistency (ACID)
Deployment platform	MUST target Kubernetes	Deployment files are the team's responsibility - not the k8s cluster IaC itself
Billing	MUST support Signicat Billing	evaluate micro-billing

6.1.2 Application

Non-Functional requirements	Description	Comment
Programming language	MUST be Java (latest stable with preview enabled, which will close some of the gap between Kotlin and older Java)	PoC implemented in Kotlin. Adding Kotlin to Signicat Green Stack's set of languages was discussed in a technology alignment meeting. (Languages are per stack, not per team or project.)
Packaging	MUST release containers	Build as a Spring boot or Quarkus application.
Versioning	MUST support versioning for continuous deployment	Defined in Versioning for Continuous deployment Deployment at Signicat
HTTP API	MUST be REST	
Signicat rest API guidelines	MUST implement the Signicat rest API guidelines	https://signicat.gitlab.io/architecture-group/rest-api-guidelines/introduction
Unit test code coverage	MUST be above 90 percent	
Linting	MUST respect linting	TODO, Options: point to a checkstyle.xml point to a sonarcloud config Which to rely upon?
Sonarcloud	MUST publish to Sonarcloud at least on each Sonarcloud merge to master	Integrate with Sonarcloud
Snyk	MUST have no vulnerabilities	Integrate with snyk.io
API Documentation	MUST offer a swagger/openapi definition	Make sure that documentation is good as https://beta.developer.signicat .com/ uses swagger
Metrics	MUST expose metrics	See existing Java apps for relevant standard metrics
Logs	MUST log to stdout MUST integrate with centralized logging MUST support deleting all logs for a given customer	

6.1.3 CI/CD

Non-Functional requirements	Description	Comment
Build and deployment	MUST be built and deployed with gitlabci	
Merge to master	MUST be merged to master by employees	Take advantage of gitlab's support for only allowing Maintainers to merge Developer's PR Consider a long lived developer branch that the students use as "their" master.

6.1.4 Security

Non-Functional requirement	Description	Comment
GDPR compliance	MUST be compliant with GDPR	Among other things, do not log things that contain PII.
OIDC	MUST secure endpoints with OIDC	TODO, do we require OIDC on all endpoints
Tenant isolation	MUST design system to guarantee good tenant isolation MUST be safe to remove tenants from live systems	Several isolation model exist at deployment level: 1 set of service 1 database. That's what signicat green traditionally does. in this case software is responsible of isolation 1 set of service n database. Giving one database per customer would be great in terms of isolation (when customer leaves we drop the whole db) M set of services N database.
Encryption in transit	MUST use TLS in exposed APIs	
Encryption at rest	MUST make use of encryption at storage backend	Multi tenancy design can lead to discussions relative to one set of encryption keys per database Key management is hard Explore support in managed services

6.2 Front-end

6.2.1 Infrastructure and dependencies

Non-Functional requirements	Description	Comment
Deployment platform	MUST target Kubernetes	Deployment files are the team's responsibility - not the k8s cluster IaC itself

6.2.2 Application

Non-Functional requirements	Description	Comment
Programming language	MUST be Javascript/CSS/HTML	Decide specific JS framework.
Packaging	MUST release containers	
Versioning	MUST support versioning for continuous Defined in Versioning for Continuous deployment	Defined in Versioning for Continuous deployment Deployment at Signicat
Unit test code coverage	MUST be above 85%	
Linting	MUST respect linting	ESLint
Sonarcloud	MUST publish to Sonarcloud at least on each merge to master	TODO, does it makes sense?
Snyk	MUST have no vulnerabilities	TODO, does it makes sense?

6.2.3 CI/CD

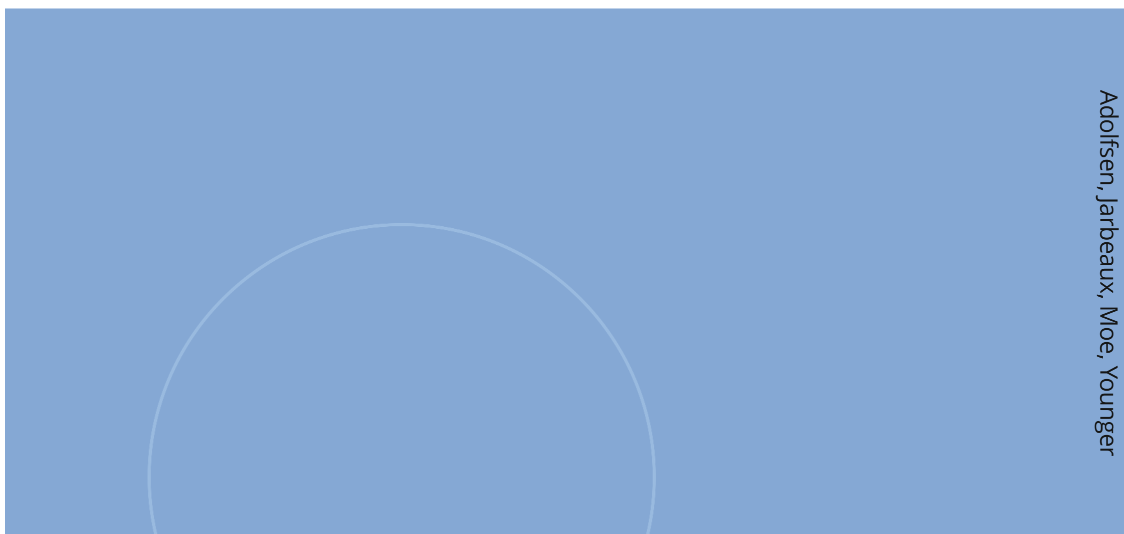
Non-Functional requirements	Description	Comment
Build and deployment	MUST be built and deployed with gitlabci	
Merge to master	MUST be merged to master by employees	Take advantage of gitlab's support for only allowing Maintainers to merge Developer's PR Consider a long lived developer branch that the students use as "their" master.

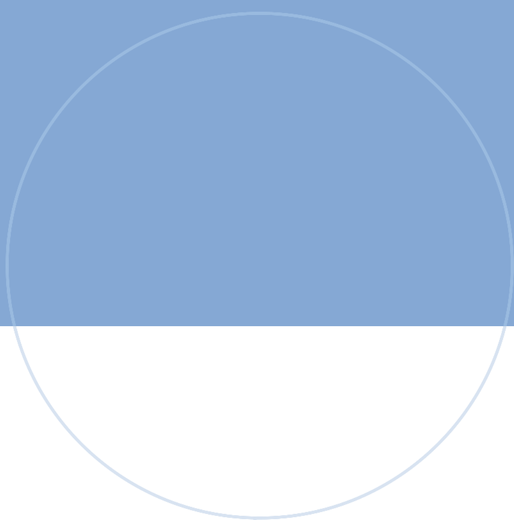
6.2.4 Security

Non-Functional requirements	Description	Comment
GDPR compliance	MUST be compliant with GDPR	
WCAG 2.1	MUST support WCAG 2.1	TODO, which standard to support?

8.5 Process Document

Note: Attachment delivered separately.





Adolfsen, Jarbeaux, Moe, Younger

Digital Evidence Management

NTNU
Norwegian University of
Science and Technology