Degree Thesis

Bachelor's Programme in Computer Science & Engineering, 180 credits

REST API & Mobile Application

Computer science and engineering, 15 credits

Ludwig Nord, Marcus Grimberg

HALMSTAD UNIVERSITY

Abstract

DH Anticounterfeit, a company working against counterfeit products, has a mobile application that lacks a proper back-end system to handle and store DH Anticounterfeits data. There are plenty of ways to go about to fix this problem, but this thesis is focused on the development of a REST API that will be replacing the non-existent system. The main goal is to create the API and run it within a containerized environment using Docker. The whole system should run on Amazon Web Services, that will be working as a server to keep the system going at all times. Nowadays, security is important due to plenty of ways to attack a system, which is why security and authorization will be a high priority. The result will be a well-functioning, high security system that uses MongoDB database to store and process data in a secure way.

Sammanfattning

DH Anticounterfeit, ett företag som jobbar mot förfalskning av produkter, har en mobilapplikation som saknar backend-system för att hantera och lagra DH Anticounterfeits data. Det finns massor av olika sätt att hantera detta problem, men detta examensarbete fokuserar på utvecklingen av en REST API som kommer ersätta det icke-existerande systemet. Målet är att skapa en API och köra den i en containeriserad miljö med hjälp av Docker. Systemet ska köras på Amazon Web Services, som kommer arbeta som en server för att systemet ska vara igång hela tiden. Nu för tiden så finns det massor av sätt att attackera ett system, vilket är varför säkerhet är väldigt viktigt, och varför säkerhet och autentisering kommer vara en hög prioritet under projektet. Resultatet kommer vara ett välfungerande system med hög säkerhet, som använder sig av en MongoDB-databas för att lagra och bearbeta data på ett säkert sätt.

Contents

List of acronyms

- **API** Application Programming Interface
- **AWS** Amazon Web Service
- **BaaS** Backend as a Service
- **CPU** Central Processing Unit
- **EC2** Amazon Elastic Compute Cloud
- **ECS** Amazon Elastic Container Service
- **GDPR** General Data Protection Regulation
- **HTTP** Hypertext Transfer Protocol
- **IAM** Identity and Access Management
- **IP** Internet Protocol
- **ISO** International Organization for Standardization
- **JSON** JavaScript Object Notation
- **ORM** Object-Relational Mapping
- **RAM** Random Access Memory
- **REST** Representational State Transfer
- **S3** Amazon Simple Storage Service
- **SaaS** Software s a Service
- **SSD** Solid State Drive
- **SQL** Structured Query Language
- **TCP** Transmission Control Protocol
- **TLS** Transport Layer Security
- **UDP** User Datagram Protocol
- **UI** User Interface
- **VM** Virtual Machine
- **VoIP** Voice over Internet Protocol
- **WSGI** Web Server Gateway Interface
- **XML** Extensible Markup Language

1 Introduction

DH Anticounterfeit AB [1] started in 2016 as a carve out from the IT consultant company DH Solutions (Datahalland) which has been in the IT industry since 1972. Today DH Anticounterfeit provides a web based case management system specialized in managing anti-counterfeit cases. The current platform is built in a monolithic way.

The web-based case management system is delivered in the form of a "Software as a service" (SaaS) solution in the Amazon Web Service (AWS) cloud. Information gathering and the security around that is important for DH Anticounterfeit, therefore DH Anticounterfeit is ISO 27001 (information security) certified.

The clients using the web based system today are from different market verticals but they have the same goal to make sure counterfeits of their products get off the streets and not in the hands of their consumers. Since the counterfeit usually are of poor quality, it could cause physical harm to the end consumer, but also loss of profit for the company.

DH Anticounterfeit has an investigation mobile application (app) that is not on the market today. DH Anticounterfeits goal is to offer this app as a product to their customers. Today the app is platform independent and built using Xamarin (C#). Xamarin helps, as stated by Microsoft to "Deliver native Android, iOS, and Windows apps with a single shared .NET code base."[2]. The big problem that they have is that there is no back-end system to support the application. Today, the application uses email to send data, which causes problems, such as retrieving and storing the data.

What the company is expecting from this project is a system where data is being transferred to a database, and also allows data to be retrieved again. Since security is important, authentication on both the database and the API is highly requested.

For this project, some of the decisions we have to make depend strictly on our preferences, and some decisions are made by the company, which makes it hard to be impartial and subjective in product design and evaluation.

1.1 Purpose and Goal

The purpose of this project is to develop a back-end solution for DH Anticounterfeit's mobile application. Since DH Anticounterfeit is ISO certified, this is something that has to be taken into consideration when designing our new back-end. For the mobile application to be able to use the back-end system, the application has to be slightly modified.

1.2 Delimitations

To keep the project within reasonable size, some boundaries/delimitations has been set as follows;

- Existing mobile app will be used instead of developing a new one;
- The mobile app should be compatible with android in focus;
- The system will not comply with all ISO 27001 requirements;
- The system will not include backup of data contained inside the database;
- No real-time database will be implemented.

1.3 Requirements

DH Anticounterfeit has set a list of requirements that this project has to fulfill. Below is a list of these requirements.

- A REST API must be used to communicate with the server from the mobile application;
- The REST API must follow a standard, preferably the OPEN API standard;
- Docker must be used as a platform for the REST API;
- As a database, MongoDB has to be used.

1.4 Main Questions

The main questions that are supposed to be answered for the dissertation are described below.

- How to design a REST API with the ISO 27001 standard in mind?
- What modifications on the current mobile app have to be made to work with the REST API?
- What makes a REST API better than the current method?

2 Background

The amount of systems and applications on the market just keep increasing, and to get a stable system where user information, and other kinds of data are important, a solid back-end system is required. Many factors come into play when deciding whether a system is good or not, and it all depends on what the system is going to be used for. One system might be developed with only performance in mind, where the speed of the data transfer is important. Another system might prioritize security, where the users integrity and privacy is the most important part.

2.1 Related Work

2.1.1 Storing data

To store the data of the users, or data about a system, the most commonly used option is a database. There are many types of different databases to use, such as a document database, relational database, network database and so on. Instagram, which handles over 1 million requests per second, uses PostgreSQL and Cassandra [3].
PostgreSQL is an object-relational database which uses multi-version concurrency control (MVCC). This allows for much better performance, which a 400 million users-per-day applications require [4].
Cassandra is a cloud-oriented database, which is designed to store large amounts of data from several servers, while still offering consistent data and high availability [5]. It is also easily scalable, which allows the developers to expand their infrastructure, and handle more users.

2.1.2 Authentication and Authorization

Authentication and Authorization are widely used across multiple services. Authentication is mostly used in form of a username and a password to ensure that the user is legitimate before authorizing that user to access different resources. Resources that are accessible through HTTP (HyperText Transfer Protocol) give easy access but opens up for a lot of security issues due to it being widely open to the public. A solution to the security issues using HTTP is HTTPS (Hypertext Transfer Protocol Secure) which transfer data through a secure communication channel. This is what a REST API can take advantage of to establish a secure connection between a server and a client [6].

2.1.3 Backend as a Service (BaaS)

Mobile usage has increased significantly lately. According to GSMA Intelligence [7] there were 5.1 billion people subscribed to mobile services by the end of 2018 around the world. This creates popularity when it comes to mobile applications. As the mobile usage grows, so does the cloud computing. The reason cloud computing grows parallel to the mobile usage is to overcome limitations of the

mobile computation. This opens up for more advanced applications. With the demand of cloud computation a new type of service emerged called Backend as a Service (BaaS). What this allows is for developers to link their apps to cloud storage, user authentication, user authorization as well as push-notifications [8]. An example of this is Firebase, made by Google [9]. A company could use a BaaS instead of building an entire solution from scratch. This saves time but will add limitation on functionalities and will give control of a back-end-solution to other companies, which can be a problem for some.

3 Theory

3.1 Network connections

To transfer the data over a network, different protocols can be used, depending on the data that is transferred, and its criteria. This part is vital for the project to work as intended, since a back-end system needs to transfer data correctly, and with the proper security.

3.1.1 HyperText Transfer Protocol (HTTP)

The world wide web (www) is all about the communication between the client and servers. To be able to communicate, the clients and the servers send HTTP requests, and receive HTTP responses [10]. An example of a scenario where a client and server communicates are described below.

1. A clients sends a HTTP request to the web.
2. An web server receives the HTTP request.
3. The server processes the request by running an application.
4. The server sends an HTTP response to the client.
5. The client receives the HTTP response.

HyperText Transfer Protocol Secure (HTTPS) is most often used, which is the same as HTTP but with security. This method encrypts the data using Transport Layer Security (TLS). This allows for a more secure transfer of data, and is used by most sites on the internet.

3.1.2 Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP) [11] runs over Internet Protocol (IP). It is a reliable way of transferring data packets from the source to the destination. The data packets are guaranteed to reach the destination. On the other hand, the data packets are not guaranteed to reach the destination in the same order. The solution to this problem is to number the data packets so it can be ordered upon arrival. TCPs area of use is for example when transferring large files. TCP uses re-transmission to guarantee packet delivery and maintain the communication stream for larger amount of data.

You need the reliability that all of the packets reach the destination to not end up with a corrupted file due to some missing packets. Loss of packets using TCP is not that vital due to the capability of re-transmitting packets.

3.1.3 User Datagram Protocol (UDP)

User Datagram Protocol (UDP) [11] is a lightweight alternative to TCP. UDP also run over the IP and is unlike TCP a ”unreliable” protocol in the sense

that you can not ensure packet delivery. The result of small amounts of missing packets will be a lower resolution of the audio or video. UDP is a good alternative for example for video streaming but not that persistent when it comes to ensure the packets arrival to the destination.

3.2 Hosting the system

In order to keep a system running at all time, a server is needed. A server job is to provide functionalities to different clients, also known as the users. This is necessary in this project, since the back-end part of the system needs to be running 24 hours a day, so that the application is usable whenever it needs to be.

3.2.1 Amazon Web Services

Amazon Web Services (AWS) is a cloud services platform. It offers data storage, compute power, content delivery and many other functionalities. AWS helps businesses to build sophisticated systems with increased reliability, scalability and flexibility. Examples of companies that uses AWS are Netflix, who delievers billions of hours of content to their customers, and Airbnb, who stores backup and static files on AWS, including over 10 petabytes of user pictures [12]. AWS offers different services, for example Amazon S3, Amazon EC2, and Amazon ECS.

- **Amazon Elastic Container Service (ECS)** is a scalable orchestration of high-performance containers that allows the developer to run and scale applications on AWS with ease. With ECS, the developer will not need to install and operate containers, or manage and scale a collection of virtual machines. ECS gives the developer a possibility to use simple API calls, to launch and stop Docker applications, access features such as security groupes, IAM roles and much more, and it allows the developer to query the state of the application [13].

- **Amazon Simple Storage Service (S3)** is an object storage service, which offers performance, security, data availability and scalability. This allows for all kinds of use, from store and protect data, to big data analytics. S3 is described as an easy-to-use management system, that lets you fine-tune the configurations of access controls, and organize your data [12].

- **Amazon Elastic Compute Cloud (EC2)** is a web service that offers complete control of the computing resources that are being used, and runs on Amazon's computing environment. EC2 allows for quick scale capacity, and reduces the time it takes to boot new server instances. EC2 is also very inexpensive, due to the fact that you only pay for the capacity that you actually use [14].

3.3 Back-end

A back-end is used as a interface between the host computer and the database. In simpler terms, the back-end system receives data from the front-end system, in our case the mobile application, and handles the data in different ways, for example a REST API, or using GraphQL, as described below[15]. Different databases can be used as well, which is described in section 3.3.3.

3.3.1 RESTful API

Application programming interfaces (APIs) is used for client programs to communicate using web services. A RESTful API is a form of web API, which is the face of a web service that directly listens and responds to the requests from the user, most often using HyperText Transfer Protocol Secure (HTTP(S)) [16]. The name is short for REpresentational State Transfer, and it contains guidelines to make the API RESTful. It uses client-server architecture, which is described above. This allows for a more portable user interface between multiple platforms, and improves the scalability of the system. A RESTful API is also stateless, which implies that each request from the client has to contain all the necessary information, to understand the request. This makes so that the session state is kept entirely on the client side. The most common requests is POST, GET, DELETE and PUT. These request allows the system to perform different tasks depending on what HTTP method being sent. Some tasks can be to put data into a database, retrieve data from the database, delete data, and update data. An example of a GET request would be:

```
https://api.instagram.com/v1/media/search?lat=41.87\&lng=87.62\&distance=20
```

This request uses instagrams API, and returns instagram posts at a certain location.

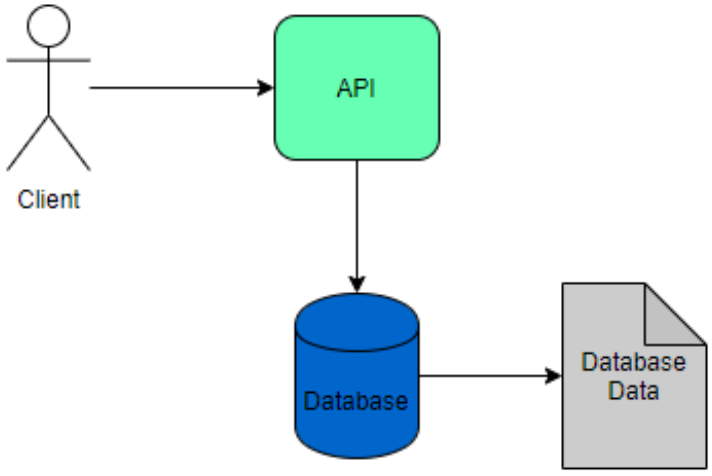

Figure 1: General architecture of a RESTful API system

3.3.2 GraphQL

GraphQL is an open-source query based language for APIs. It is developed by Facebook, and used by many big websites, such as Facebook and Github. As stated by Facebook, GraphQL [17] is "a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.". GraphQL is a query lanugage (QL) in which you can build APIs as strongly-typed schema instead of multiple REST endpoints. This reduces the amount of requests to gather all required data [18]. See Figure 4 for a simple GraphQL schema with three different types, Query as the Entrypoint, Post and Author. Clients write a query for what they want and get back exactly what they asked for, nothing more. In Figure 2 is an example of a request in GraphQL which ask for the name and email from user with id 2 which produces the result seen in Figure 3.

```
query {
  user(id: 2) {
    name
    email
  }
}
```

Figure 2: A GraphQL request

```
{
  "user": {
    "name": "John Doe"
    "email": "jogndoe@test.com"
  }
}
```

Figure 3: The result from Figure 2

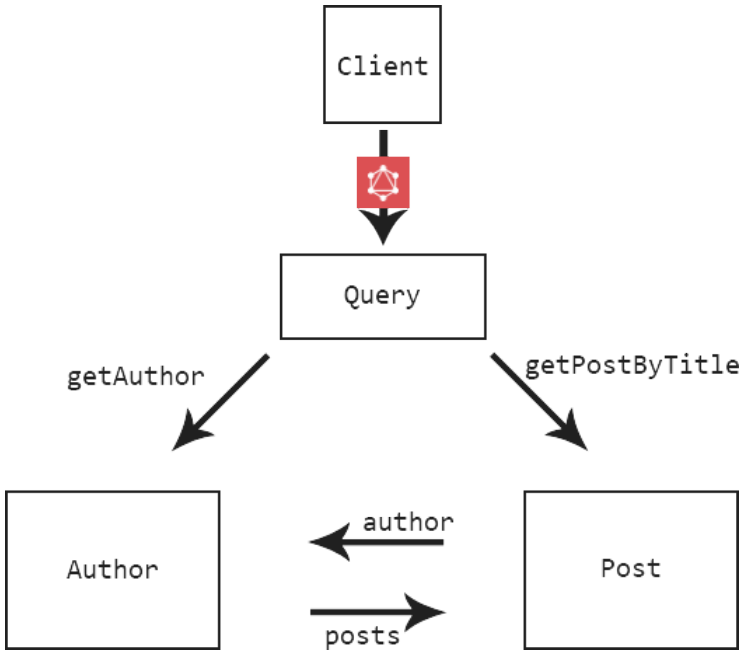

Figure 4: Example of a GraphQL schema

3.3.3 Database

SQL and NoSQL databases are different types of databases that are commonly used today. Below both of these types of databases are described. Both of these types of databases would be valid options for this project. However, there are some key differences, such as the way they scale, and whether they have a predefined schema or not.

- SQL is based on the relational model and is used to query and manipulate data in a database. It enables multiple users to access the same data simultaneously without disrupting each other. SQL has the ability to carry out joins which combines rows from multiple tables. SQL is also able to perform complex aggregation functions where the values of multiple rows are grouped to display a single value. See Figure 5 for an example of a

one-to-many relationship between a Student and a Course [19]. An SQL database scales vertically, meaning that you can increase the performance of the database, by for example upgrading CPU, RAM or SSD. This can be a disadvantage, since you can only upgrade so much. This type of database also uses a predefined schema, which means that one has to structure the data before working with it, and all data must follow the same structure. This makes for a difficult change in structure if necessary. An example of a relational database that uses SQL is MySQL [20].

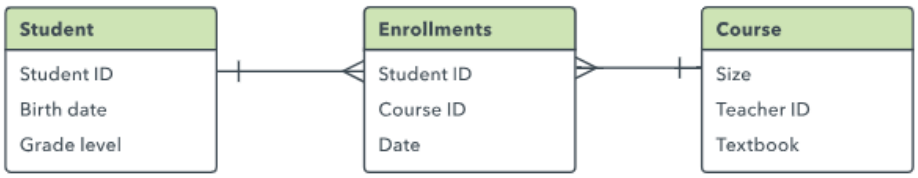

Figure 5: Example of SQL structure

- NoSQL, unlike SQL stores key-value pair as the data. These types of databases is usually in the form of XML or JSON structure. In Figure 6 you see an object using XML format and in Figure 7 the same object is created using JSON structure [21]. NoSQL databases typically scale horizontally. This means that in order to increase performance, more servers could be added. This gives NoSQL a clear advantage over SQL, since it can become more powerful and larger. NoSQL does not use a predefined schema, which allows for easy change in the structure of the data. An example of a document-oriented database is MongoDB, which uses JSON structure to store data.

```xml
<employees>
  <employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
  </employee>
</employees>
```

Figure 6: Object in XML format

```json
{
  "employees": [
    {
      "firstName": "John",
      "lastName": "Doe"
    }
  ]
}
```

Figure 7: Object in JSON format

3.3.4 Docker and Containers

A container main objective is to remove the struggle of getting applications to run in different environments. A Container packs the applications code and all the dependencies specified, to simplify the possibility to run the application in multiple different computing environments [22]. In this project, it allows for an easy transition when deploying all components to a server.

The main objective of docker is to make the deployment of applications inside containers automated. This enables faster boot-ups, since it is more lightweight than regular containers. Docker uses container virtualization technology, instead of using a regular Virtual Machine (VM), since VM is a fairly heavy-weight computer resource [23]. Docker contains 3 components, which are described below [24].

- Docker image are used to create a Docker container, and is a read-only template. For example, an image can contain an operating system and other web applications.
- Docker Registries are used to store Docker images. It can be either public or private, to download and upload Docker images.
- Docker Containers are holding the necessary bits for a application to run, similar to a directory. A container is visualized in Figure 8.

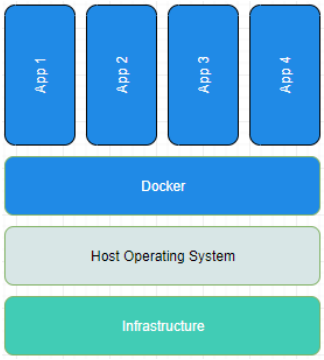

Figure 8: Architecture of a Docker container

3.4 Front-end

A front-end is used as an interface between the host computer and the external inputs, for example terminals, computer networks etc. The front-end is pretty much what the user sees, and sends the data to the back-end [15]. This is highly relevant for us, since we have the choice to further develop the existing application, or create a completely new one.

3.4.1 Native development

Native development means that the application only works on a specific platform. This type of development typically leads to better user experience, and better performance due to better communication between the device, and the code. However, this may be unnoticeable in a well-built system. The cons with native development is that it takes so much time, and costs more money, since

other applications have to be developed for the other platforms [25]. Examples of native development languages are Swift for iOS, and Java for Android.

3.4.2 Cross-platform development

Cross-platform development means that the application works on multiple platforms, for example iOS and Android. This leads to faster deployments for all platforms, and shortens the development time, which in turn saves money. However, it may cause some performance glitches and user experience issues, since features on one platform, might not be available on another platform [25]. An example of a cross-platform development language is Xamarin. It uses C#, and produces applications for iOS and Android.

3.5 Programming languages

When developing both front- and back-end, multiple programming languages are available. The choice of programming language comes down to official support from each tool-provider, such as Docker and MongoDB. Below is a list of different programming languages that are related to the work of this thesis.

3.5.1 Back-end

Python is the recommended language to develop the REST API in. It allows us to use Swagger, which was recommended to us in order to follow the OPEN API standard. Swagger helps designing a structured REST API, and uses a tool which gives a clear visualization of the data that is being handled.

- **Python** is a high-level, object-oriented programming language and considered an easy language because of the syntax. The fact that Python uses an interpreter instead of a compiler, gives it the possibility to skip the compilation step, which makes the edit-test-debug cycle very fast. Python is often used for scripts and to connect existing components together [26].

3.5.2 Front-end

The following languages is what is being considered for the development of the user application. If we plan to start over and create a new application, we have to debate whether to use cross platform, or native development, which is described in Section 3.4. However, if we decide to continue with the existing application, we have to use C#, which is described below.

- **Swift** is an open-source programming language made by Apple. It is built with performance in mind, and with a simple syntax for easy and sufficient code. This is an example of a language used for native development, since it is used for iOS, macOS, watchOS, tvOS and Linux applications [27].
- **Java** is a high-level, class-based, object-oriented programming language. It is commonly used for the general purpose, and is considered an easy-to-learn programming language [28].

- **C#** is an object-oriented programming language that runs on the .NET Framework. It is used to create a variety of different systems, such as web services, database applications, client-server applications and much more [29]. Its syntax is very similar to Java, which makes for a smooth transition for an experienced Java developer. C# is an example of cross-platform development, since it is used when developing applications in Xamarin, Unity and so on.

3.6 Security

There are many different ways to make a system relatively safe from attacks, though a bulletproof system is not possible at the moment. There are security measures to be made in the code to prevent data from being leaked during cyber attacks, for example man-in-the-middle attacks.

Man-in-the-middle attacks can be described as a third party that is actively eavesdropping between connections. By doing this, the attacker can for example get the information that is transferred over the connection, or manipulate the data before it is sent to the other connection, and receive desired data [30]. A way around this would be to encrypt the data. By doing this, the attacker can not read the data without the necessary private key. However, the attacker can still manipulate the data. This will be noticed by the receiver though, and the connection can be shut down.

3.6.1 ISO Standards

The International Organization of Standardization (ISO) is an organization that creates documents that provide guidelines to ensure that a service, product, material or process is suitable for the intended purpose. An example for an ISO standard that is used for data security is ISO 27001.

ISO 27001 describes how a company should behave in order to keep their data safe. It goes all the way from physical security, where an attacker might see some information when at the office, to software security, and how to handle data being lost [31]. ISO 27002 are recommendations on how to interpret and can help to meet the requirements that are in 27001, ie it is 27001 that you get certified against and which contains all the requirements.

4 Method

The project model used during this thesis is Linköpings Projektstyrningsmodell (LIPS) that is structured as described below:

- **Before:** The first phase consist of planning the project and collecting theoretical information that is needed. All the preparation work is made during this phase, for example creating the specification, project plan and the time plan but also getting familiar with new software that should be used.
- **During:** During the second phase, the actual work will be carried out and the focus is on implementing and testing the ideas developed during the previous phase.
- **After:** It is during the third phase the work will be analyzed and verified. The report made for the project should also be done at this phase that includes result, conclusion, discussion but also improvements that can be made.

4.1 Transfer data

4.1.1 Internet Protocol (IP)

Transferring data between the server and the application, both methods could be used as described in section 3.1.2 and 3.1.3. There is a big difference in reliability. UDP would work without a connection setup and also not re-transmit lost packages. This introduces less delays which would be perfect for services like video streaming and/or Voice over IP (VoIP) but not for transmitting large files or data that has to be the same when leaving the server to arrival of the client, or vice versa.

Taking the above into consideration the conclusion was that TCP is most suitable for this particular application. TCP will setup a formal connection between two sides. Whenever a packet is sent from one side there will be sent a packet from the other side confirming the arrival of the packet. Even though this introduces delays it also introduces error correction which will give the reliability needed for this kind of application.

4.1.2 Transfer data over TCP

As described in section 3.3.1 and 3.3.2, RESTful & GraphQL is usually a form of a web API which uses HTTP to transfer data. Since HTTP runs over TCP this satisfies the need for a reliable connection as specified in section 4.1.1. Both of these methods would be a possible solution for this project.

If uploading large files like videos is a requirement, switching to another dedicated service handling the upload would probably be advised. This approach is

forced upon when using GraphQL and can be overkill for uploading images and could enlarge this project so that the deadline will pass. Based on these factors a REST API is chosen to be designed and developed as the way of transferring data between the server and clients.

4.2 Environmental setup

4.2.1 Hosting platform

When choosing an Amazon Web Service, one realizes that multiple services match the requirements for some task. Section 3.2.1 describes different instances of an Amazon Web Services that could be used for this thesis project. Amazon Elastic Compute Cloud (EC2) is simply a remote virtual machine running some server OS (Server Operating System). This allows for a server without much constrains and can be developed for different purposes. If one were to distribute a system with multiple EC2 instances running together, the complexity of a system greatly rises. Choosing a single t3.micro (as Amazon refers it to) EC2 instance is completely free for a year, as long as the instance does not exceed the free tier limits. The free tier limit is one container. What a container contains is described below in this section.

ECS on the other hand, is in simple terms a manager. A manager that manages a cluster of EC2 instances and each EC2 instance can but is not limited to act as a Docker host. This allows for easy to run and scalable containerized applications. Since you distribute the entire solution into multiple instances the general performance of the system will increase, since the workload is not focused on one EC2 instance as in the above example. This solution does not fall in the category of free-tier services provided by Amazon.

With all the above information taken into consideration the final conclusion was to choose a service that falls into the free-tier category, due to the project being tight on a budget and no money could be spent on services that would cost money. EC2 t3.micro is the instance chosen for this project due to it being free and would also give the ability to migrate to ECS in the future if DH Anticounterfeit would like to. A T3.micro instance provides 2 virtual CPUs (vCPU). Each vCPU is run within a thread on an Intel Xeon core which gives each instance a clock-rate of 2.5GHz [32] and will be enough for this project. In Section 3.2 the importance of a service running 24 hours a day is described. For a single EC2 instance, AWS promises an hourly up-time percentage of 90% minimum which is more than this thesis project needs [33].

4.2.2 REST API using a web framework

For building the REST API a web framework named Flask will be used. A web framework is a collection of different packages or modules that makes it easier to build web applications. These frameworks make it easier to reuse code for common HTTP operations like GET, PUT and POST. Any web framework for Python could be used for this project. Some Web frameworks tend to add a lot

of extra functionality that will not be necessary for this project and has a deep learning curve and is therefore discarded. Django is one of these frameworks that has a lot of functionalities like Object-relational Mapping (ORM) which is orientated towards SQL-databases.

Building a REST API with Flask is easy due to the URL routing. Adding one line above a method turns that method into a callable object when visiting the matching URL as shown in Figure 9. These methods with their own path defined, makes up the REST API.

```
from flask import Flask
app = Flask(__name__)

@app.route("/api/hello")
def hello():
    return "Hello World!"
```

Figure 9: Visiting path '/api/hello' displays a text equal to 'Hello World!'

Defining all paths and method to comply with the OPEN API standard, which is the standard chosen for this thesis, can be a hassle. Solely for that purpose an editor called Swagger is used. By defining all paths and their respective HTTP operations and parameters, Swagger can then generate a skeleton for different servers that complies with the OPEN API standard. Figure 11 shows how a path is structured in Swagger Editor. This example is for path '/token'. The GET operation is defined to have two parameters, a username and a password, both of which are required and is of type String. The response is also defined for the status code 200. Generating a Python web framework for OPEN API standard 3.0 is not supported in Swagger Editor. A python generator from a user named guokr on Github where used to generate the skeleton from the Swagger specification[34]. In Figure 12 are all the directories and files generated by the python generator. __init__.py is the root directory that creates and runs the Flask app and imports the v1 package. __init__.py are files that get executed when they are being imported as a package. __init__.py in the v1 folder imports the routes.py file which contains all paths and their respective .py files and adds them as a resource to the flask-app. validators.py validates a request sent to the Flask app according to the schemas.py. Schemas.py contains schemas based on what was entered in the parameters and response tag in the Swagger Editor. An example is the response tag in the Swagger Editor as shown in Figure 11 that correspond to the following code in schemas.py as seen in Figure 10.

```
filters = {
    ('token', 'GET'): {200: {'headers': None, 'schema': None}},
}
```

Figure 10: How a response tag is converter into Python code

```yaml
paths:
  /token:
    get:
      tags:
      - token
      summary: Ask for an access token
      operationId: get_token
      parameters:
      - name: username
        in: query
        description: The username
        required: true
        schema:
          type: string
      - name: password
        in: query
        description: The password
        required: true
        schema:
          type: string
      responses:
        200:
          description: successful operation
          content: {}
```

Figure 11: How a path is created in Swagger Editor

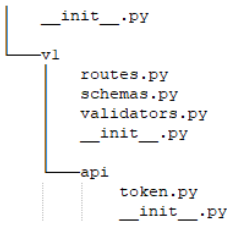

Figure 12: Structure generated from the Swagger specification from Figure 11

4.2.3 WSGI Server

Traditional web servers have no way of running a Python applications. The WSGI (Web Server Gateway Interface) is just an interface specification on how servers and applications communicates. These specifications are described in the PEP 3333, which can be found on python.org. A WSGI Server is simply taking requests from a client and passes that request to the application. The application then returns a response to the WSGI Server which in turn sends this to the client.

Any of the available WSGI Servers could be used for this project. This thesis uses the uWSGI project to achieve the things described above. uWSGI is fast and simple to get running with Flask. After installing uWSGI an initialization file has to be created. The initialization file for this project is really simple and has predefined where the module is located. The module in this case is the wsgi.py file to be executed on start-up of uWSGI. wsgi.py is simply a .py file in which the Flask-app is started.

4.2.4 NGINX as a reverse proxy

NGINX [35], is as they describe it "a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server". While NGINX has different use-cases this project is only interested in having NGINX as a reverse proxy. A reverse proxy direct client requests to the appropriate backend server, for this thesis that server is our WSGI server as described in Section 4.2.3.

Installing NGINX is really simple, using the following commands executed in a shell will install NGINX on Ubuntu 16.04 (Xenial) which is the OS this project is using.

- deb http://nginx.org/packages/ubuntu/ xenial nginx
- deb-src http://nginx.org/packages/ubuntu/ xenial nginx
- sudo apt-get update
- sudo apt-get install nginx

After installation, one has to make sure NGINX is running. This is done by executing the following command into a shell: **service nginx status**. This should return a bunch of information, including **Active: active (running)**. If not, simply executing **service nginx start** should start NGINX. Next step is to change the nginx.conf file which is located inside of the NGINX folder which usually is located inside **/etc/nginx/** directory. The addition to the nginx.conf is shown in Figure 13 and should be placed within the http{} brackets.

```
server {
    listen 443 ssl;
    server_name YOURDOMAIN;

    ssl_certificate     /home/ubuntu/python-flask-uwsgi-docker-https/instance/TLSCERT.crt;
    ssl_certificate_key /home/ubuntu/python-flask-uwsgi-docker-https/instance/TLSKEY.key;
    ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers         HIGH:!aNULL:!MD5;

    location / {
        proxy_pass https://0.0.0.0:8443;
        proxy_set_header    Host            $host;
        proxy_set_header    X-Real-IP       $remote_addr;
        proxy_set_header    X-Forwarded-for $remote_addr;
        port_in_redirect off;
        proxy_redirect   off;
    }
}
```

Figure 13: The addition to nginx.conf

What the configuration shown in Figure 13 does is listening on port 443. Port 433 is the standard TCP port that is used for website which uses TLS. Each time a client access the domain through port 443, NGINX has to know what TLS certificate to use in order to establish a secure connection. NGINX also has to know what TLS protocols and ciphers it should allow from a client. Below all of this is the reverse proxy configuration. The configuration is set to pass all requests on port 443 to address 0.0.0.0 on port 8443. In this case, this is the WSGI Server. There are also some header information set to be the clients address and can be used within the Flask app if necessary.

4.3 Docker & structure of containers

Docker allows for a easy transition from developing on our local machines to AWS. To firstly develop on our local machines, it allows us to easier test and develop the program, to later transfer it to AWS, and it will work in the same way, which is what Docker is all about. Installing goes smoothly by running an installation file. However, the Windows version that the machine is using matters, since the most common version, Home, does not use Hyper-V. Hyper-V allows the user to run multiple operating systems, using virtual machines.

To go around this problem, the Docker Toolbox can be installed instead, which goes smoothly with an installation file.

4.3.1 Containers

The whole system is built using 2 containers. One running the whole REST API and the necessary components to transfer the data. The other container runs MongoDB, to control the data. There are many ways that the amount of containers could have been set up, and what they contain. Since we desired a simplistic and minimalistic system, 2 containers was just about right, to still have some structure.

- **MongoDB** is easily started as described in Section 4.4.1. By stating the ports, a database name, and a user with read and write permissions, it is easy for the API to connect to the database.
- **The REST API** container contains a lot more components than the other container. All these components are described in Section 4.2.2. This container is started using a "Dockerfile" instead. It works the same as the way the database container is started, but with a file instead. PyMongo, which is what is used to interact with the database in the API, has an easy way to connect to the database, by simply stating the port, database name, IP address and user.

4.4 Storing data

4.4.1 Database

To store the data from the application both SQL and an NoSQL database could be used as described in Section 3.3.3. Due to requirements from the company, MongoDB is the database that will be used in this thesis. MongoDB is compatible with docker, and the fact that it uses JSON structure, which will handle the data that the application is sending well.

Setting up MongoDB goes quite smoothly due to its compatibility with Docker. Firstly, an image has to be downloaded on Docker, containing everything needed to run MongoDB. Different images can be downloaded containing different versions of Mongo, depending on what features are needed. By running the command "Docker pull mongo", in the Docker terminal, the latest version of Mongo is downloaded, with the "latest" tag. When the proper image is downloaded, a container can be started. By running the command:
"docker run -d -p 27017:27017 --name my-db -v /Desktop/data mongo". This starts a container in detached mode (-d), it specifies the ports on both client, and server side (-p 27017:27017), the container is named "my-db" (--name my-db). A path for the container is stated (-v /Desktop/data), and the image that is being used is stated last. The path is important. It allows the database to keep the data, even if the container fails and shuts down for some reason. In order

to interact with the database, an interactive terminal can be used. When the container is running, the command: "docker exec -it my-db bash" can be used. This executes (exec) an interactive terminal (-it) using the bash shell. Now, instead of using the regular terminal, the terminal within the container is used, and the command: "mongo" can be used to access the database. Although this creates a fully functional database, some kind of authentication is necessary, so that not everyone can access the database. Mongo has a built-in access control function, which is enabled by adding "--auth" in the run command, making it: "docker run -d -p 27017:27017 --name my-db -v /Desktop/data mongo --auth". This enables SCRAM-SHA-1 authentication mechanism, and when enabled, one must authenticate themselves to access the database. It is important that the Mongo version is the same as the Mongo shell version, since different authentication methods are used in different versions. This can be checked with the command: "Mongo -- version". Before enabling authentication, a user that can log-in has to be created. When in the Mongo container, Figure 14 demonstrates how to create an admin user:

```
db.createUser(
    {
        user: "useradmin",
        pwd: "example",
        roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
    }
)
```

Figure 14: Shell command to create user

Now that a user has been created, the database can be accessed with authentication enabled. To visualize the content of the database easier, a program called Robo 3T is used. It is a free GUI for MongoDB. It allows for authentication tests, and the ability to interact with the database. Creating a user could have been done in the program, instead of using the bash shell.

4.5 Security

4.5.1 Authentication and authorization for protected resources

When all paths, parameters and responses are defined as explained in Section 4.2.2, depending on the sensitivity of information stored within a database there has to be some form of authorization. The reason for this is to restrict users to their own resources. In order to authorize a user the user has to be authenticated first. Authentication can both be username & password based or token based. Having to send username and password with every request is not very convenient. This can also be seen as a security risk, even if the request is over secure HTTP.

To protect these resources HTTP Basic Authentication were used. Instead of implementing this from scratch the FlaskHTTPAuth extension can do this

easily. By adding **login_required** decorator to an endpoint, the endpoint gets protected as seen in Figure 15.

```
from flask_httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@app.route('/api/protectedResource')
@auth.login_required
def get_protected_resource():
    return {'Username': g.user.username}, 200, None
```

Figure 15: Protecting an endpoint

To allow users to access these protected endpoints the **verify_password** decorator is used. A function with this decorator will be invoked each time a protected endpoint is accessed to validate the credentials. The function takes two arguments, a username and a password then validates these inputs and then returns a True or a False whether the validations were done successfully or not.

As described earlier in this section, token based authentication is considered more secure and convenient and is therefore used in this project. To be able to accommodate to both username- and token-based authentication the **verify_password** function has to validate both. This is since a user first has to login with their username and password to then get a token in return.

Figure 16 shows the implementation of the callback function **verify_password**. First the token is being checked to find a user associated with that token, if this fails we assume that there was no token provided but instead a username. A check to the database is done to find a user with the provided username. Assuming there was, the password provided is checked against the hashed password stored with the user. If both password match, the authentication were successful and the g variable provided by flask is set to be the authenticated user. This solution handles both username- and token-based authentication.

```
@auth.verify_password
def verify_password(username_or_token, password):
    from ...main import mongo
    user = User.verify_auth_token(username_or_token)
    if not user:
        db_users = mongo.db.users
        db_user = db_users.find_one({'username': username_or_token})
        if not db_user:
            print('No user found with username')
            return False
        user = User(db_user['username'],
                    db_user['password_hash'],
                    db_user['_id'])
        if not user.verify_password(password):
            return False
    g.user = user
    return True
```

Figure 16: Verify a user by username or token

When a user has been authenticated there is authorization. Authorization is done within a **login_required** decorated method. Authorization is simply done by checking the username used to authenticate with the username owning the protected resource. If both of these usernames match, the authorization was successful. If there are no match, the requests is rejected.

4.5.2 Secure HTTP / Secure Channel

As described in Section 4.5.1 authentication and authorization are important. While all of this is good, there is really no point in protecting the API if the login credentials are sent over the network in clear text. To solve this problem all requests has to go over secure HTTP or HTTPS for short. To achieve a secure connection over HTTPS one is required to get a TLS (Transport Layer Security) certificate for a domain. Theses certificates are issued by certificate authorities and usually cost money. Although most of the methods cost money there is an alternative free method provided by a company called Let's encrypt. They issues free TLS certificates trusted by every major root programs, such as Microsoft, Google, Mozilla and Safari [36]. In order to obtain a TLS certificate from Let's encrypt one has to show ownership of the domain. This was done by adding a text-file with some random string of text at a very specific path. An example is to place a text-file with the text 'X6473hvc908sda' at the following URL: http://domain.com/.well-known/acme-challenge/qFMkqbB6A60. This will automatically be checked by Let's encrypt to prove the ownership of a domain and then issue a certificate.

4.5.3 ISO Standard

As a recommendation from DH Anticounterfeit, a risk assessment will be made, in order to fulfill parts of ISO 27001. This is done in order to determine the severity of lost data. The ISO standard that is tried to be fulfilled, contains

much more than software security, for example the way a guest is escorted out of the building, instead of walking around alone. Since the subject of the thesis is software, it will contain only security breaches in this area. The risk assessment will contain the cases where a security breach is most likely to happen. The severity of the attack will be stated in numbers, depending on the outcome.

4.6 Application

The final step of this project is to connect the REST API with the mobile application made by DH Anticounterfeit AB. The mobile application is rather isolated and is not made with a back-end in mind. Since the app in its current state focuses on local storage with e-mail as a one-way backup system, there are a lot of improvements that can be made but for this project the focus is to remove the e-mailing and give the application the main functions to work, this includes the following:

- User Registration
- User Login
- Create Case
- Get Case
- Update Case
- Delete Case

To switch the emailing function with sending HTTP requests, locating the methods sending these emails has to be done. Upon finding these method the discovery was made that there are multiple different methods sending these emails, each sending different kinds of information. Although there were different methods sending emails there were only two being used. Next step were to simply locate the function calls within the app and replace those with the HTTP requests to the REST API.

Implementing user registration is straightforward, which is sending a POST request to the REST API on the following path **/v1/user**. Sending POST requests in C# is pretty simple. The RegisterAsync method shown in Figure 17 shows how to register a user with a POST request. First an object of class HttpClient provided by Visual Studio is defined. Next there is a dictionary created and two elements are added. One element contains the username and the other password which is grabbed from the User Interface (UI) of the mobile application. The conversion from a dictionary to a JSON structure is done and then sent off as a POST request to the REST API. The response from that request is saved and the status-code is a check to see if it was a successful request. If the status-code is within the range of 200-299 it counts as successful.

```
public async Task RegisterAsync() {
    var client = new HttpClient();

    var dict = new Dictionary<string, string>();
    dict.Add("username", emailFromForm);
    dict.Add("password", passwordFromForm);

    var JSONifyDict = Newtonsoft.Json.JsonConvert.SerializeObject(dict);

    var content = new StringContent(JSONifyDict, Encoding.UTF8, "application/json");

    HttpResponseMessage response = await client.PostAsync("https://domain/v1/user", content);
    HttpContent responseContent = response.Content;

    if (response.IsSuccessStatusCode) {
        //Do something if registration was successful
    }
}
```

Figure 17: A POST request to the API for creating a user

To implement the user login, a GET request has to be sent to the REST API instead. In addition to this the request has to have the Authentication header field set. This field is used to send usernames and password for a user agent, which in this case is the mobile application. This GET request is sent to the path **/v1/user/login**. Figure 18 shows the Login method which does the above. As stated previously the Authentication header field has to be set before the request is sent. This is done by setting the provided email and password from the UI as a byte array and provide that as an argument to the **AuthenticationHeaderValue** class provided by Visual Studios. After that the HttpClient is updated with the new **AuthenticationHeaderValue** object. Next step is to send a GET request together with the Authentication header field and wait for the response. If the response were successful there have also been sent an access-token by the REST API which is extracted by converting the result from JSON into a dictionary. Then a global variable is set to have that access-token since this token will be used for further requests to the REST API.

```
public async Task<bool> Login() {
    HttpClient client = new HttpClient();

    var byteArray = Encoding.UTF8.GetBytes(Email + ":" + Password);
    var Authheader = new System.Net.Http.Headers.AuthenticationHeaderValue("Basic", Convert.ToBase64String(byteArray));
    client.DefaultRequestHeaders.Authorization = Authheader;

    HttpResponseMessage response = await client.GetAsync("https://domain/v1/user/login");
    HttpContent content = response.Content;

    if (response.IsSuccessStatusCode) {
        string result = await content.ReadAsStringAsync();
        var values = JsonConvert.DeserializeObject<Dictionary<string, string>>(result);
        Settings.Accesstoken = values["token"];
        await Application.Current.MainPage.Navigation.PushAsync(new BarPage(ConstantsStorage.StartingPageNumber));
        return true;
    }
    return false;
}
```

Figure 18: A GET request to login a user

For the other functions shown in the list above, the implementation is more

or less the same. The main difference is to either send a DELETE request instead of a GET request or different data within the dictionary as shown in the POST request.

4.7 Testing

During development the methods were tested separately. In order to test the methods, Curl commands were used, which is a command-tool to transfer data over multiple different protocols, for example HTTP. An example of a curl-command would be:
"-curl -u example:example -k -i -X POST 'https://192.168.99.100:8443/v1/case". This command post a new case to the database, using authentication. Depending on the outcome, a specified status code will be returned. 200 for successful, and 400 or 404 if not, depending on the reason. The reason this command would fail could for example be wrong username or password, or database failure.

On top of the manual testing during development a more organized way of validating the system had to be made. This was achieved with a program called JMeter. JMeter provides different features to create tests in order to validate a system. See Figure 19 for a overview of a project in JMeter.

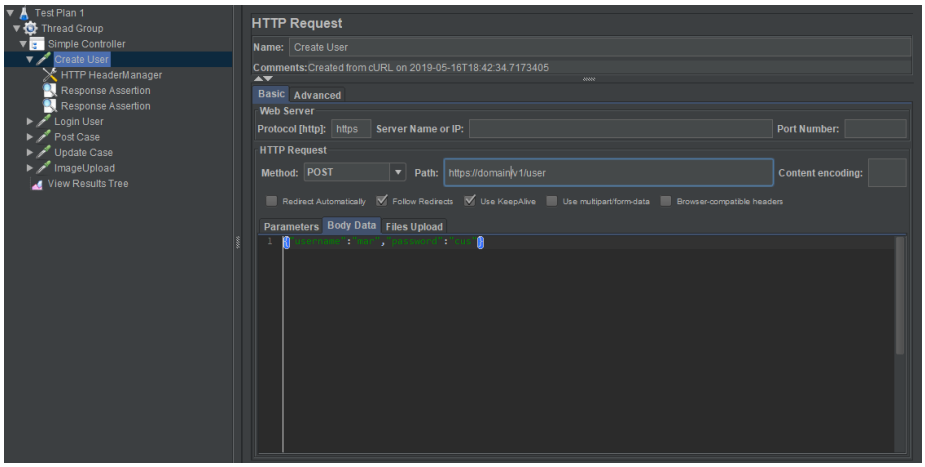

Figure 19: Overview of how JMeter looks

5 Results

In this section the result and results of the manual tests done on the system will be presented. Figure 19 shows how the entire system is setup and all connections between each component.

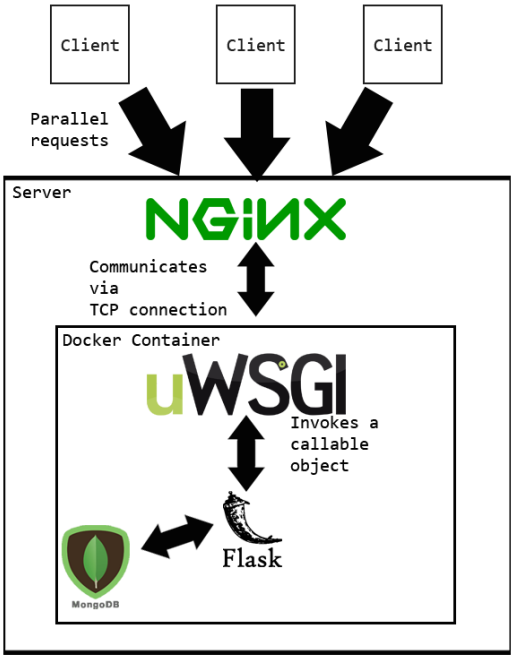

Figure 20: The system as a whole

5.1 Database & REST API

5.1.1 REST API

In Figure 21 an overview of all paths and what they do is shown using Swagger. They show what type of HTTP methods are available for each path, for example the **/case** path allows for GET, PUT and POST methods. The padlock on the right shows if the method requires a client to be authorized to access it.

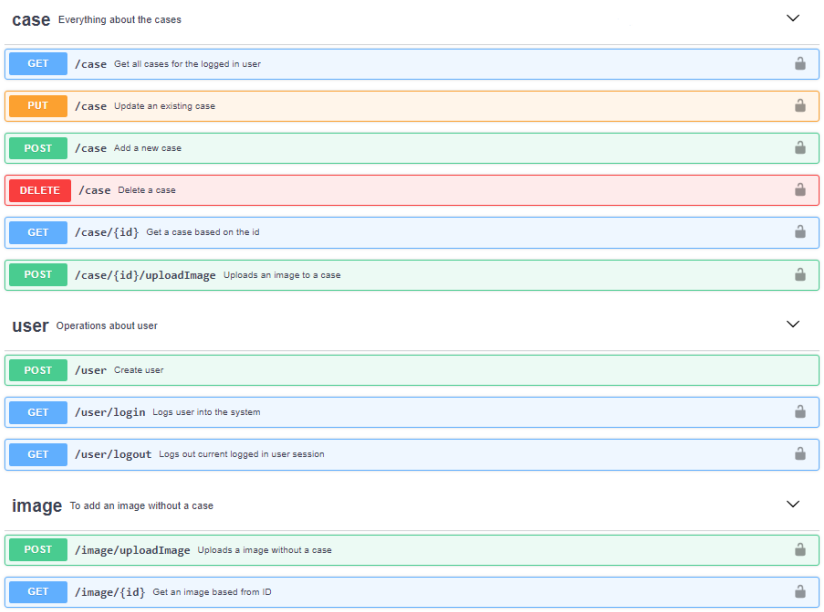

Figure 21: An overview of the REST API

Figure 24 shows a test scenario which covers a lot of the functionalities of the REST API. The table shows what HTTP method it covers and at what path it were requested from. The CURL command used to test these methods is also listed together with the response from the REST API. The test-scenario is a follows:

- A user is created with user:correctpassword credentials
- The user tries to login with wrong credentials
- The user logins with the correct credentials
- The user creates a new case with some values
- The user updates the same case with new values
- The user uploads an image

HTTP Method	URL	CURL Command	Response
POST	https://domain/v1/user	curl -i -X POST -H "Content-Type: application/json" -k \ -d '{"username":"user","password":"correctpassword"}' \ https://domain/v1/user	HTTP/1.1 200 OK
GET	https://domain/v1/user/login	curl -u user:wrongpassword -k -i -X \ GET 'https://domain/v1/user/login'	HTTP/1.1 401 UNAUTHORIZED
GET	https://domain/v1/user/login	curl -u user:correctpassword -k -i -X \ GET 'https://domain/v1/user/login'	HTTP/1.1 200 OK {"token": "eyDMe5r3A"}
POST	https://domain/v1/case	curl -i -X POST -H "Content-Type: application/json" -k \ -u eyDMe5r3A:ununsed \ -d '{"key":"value","key2":"value2"}' https://domain/v1/case	HTTP/1.1 200 OK {"_id": "5cdc6341ec63"}
PUT	https://domain/v1/case/<id>	curl -i -X PUT -H "Content-Type: application/json" -k \ -u eyDMe5r3A:unused -d '{"key":"value","key2":"value3"}' \ https://domain/v1/case/5cdc6341ec63	HTTP/1.1 200 OK {"message": "Succesful"}
POST	https://domain /v1/image/uploadImage	curl -u eyDMe5r3A:unused -i -X \ POST -H "Content-Type: multipart/form-data" -k \ -F "file=@image.JPG" \ https://domain/v1/image/uploadImage	HTTP/1.1 200 OK {"_id": "5cdc241ec64"}

Figure 22: A test scenario and its results

All of these tests were added into JMeter as shown in Figure 23. As shown in the bottom right corner of Figure 23, the entire test-case were successful.

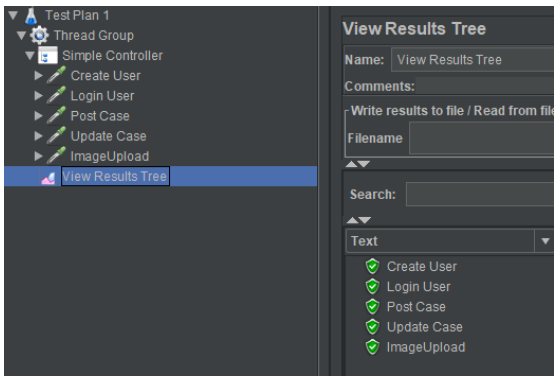

Figure 23: The above test-case constructed in JMeter

Stress-testing the entire system were done using JMeter. The test was setup to simulate 50 users performing two tasks. The first task was to login as a user and then post a case with following data: {"casenumber":"test","location":"testloc"}. These 50 tests were done over a period of 60 seconds to see the CPU usage of our EC2 instance as described in section 4.2.1. In Figure 22 the CPU usage is shown as time on the X-Axis and CPU usage in % on the Y-axis. Tests were made to see how many users the system could handle before it got overloaded. The limits we encountered was due to our personal computers not being able to send the HTTP request fast enough.

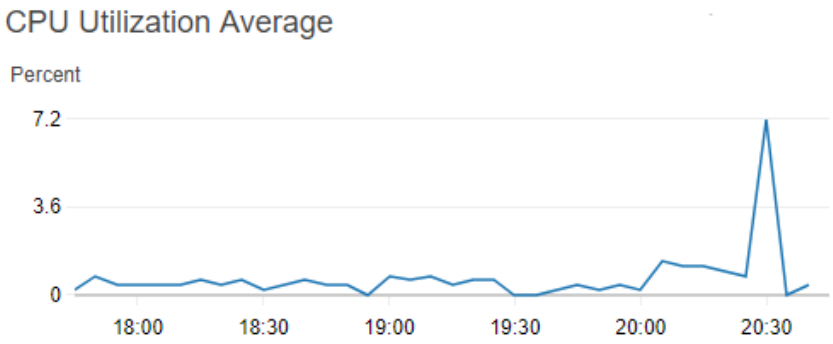

Figure 24: Average CPU usage during stress test

5.1.2 Database

The final database contains 4 different collections: cases, users, images.chunks and images.files. They all serve different functions to store the different types of objects in the system. When inserting an object into the database, an "_id" is assigned to the object, which is unique. This gives the possibility to easily update and remove objects.

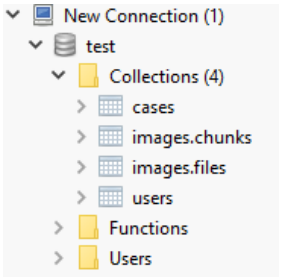

Figure 25: Structure of database

Figure 25 shows the structure of the database when seen in Robo 3T.

5.2 Security

5.2.1 Authorization for database

As described in section 4.4.1, SCRAM-SHA-1 is used to authenticate users to the database. It sets up easily and with different types of users, they can do different thing, such as readWrite, or just read.

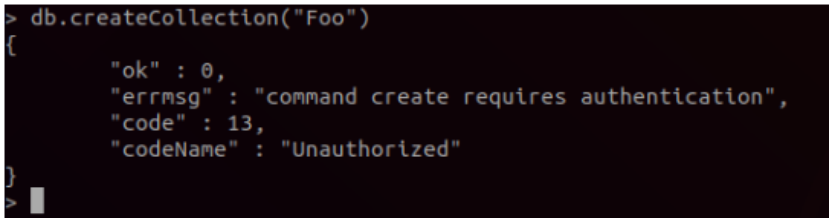

Figure 26: Unauthorized user tries to add collection

Figure 26 demonstrates what happens when an unauthorized user tries to add to the database. This indicates that the authorization for the database works. This could also be tested on Robo 3T. An attempt to connect to an authorized database in Robo 3T is shown in Figure 27.

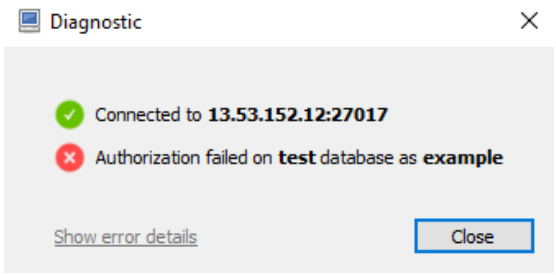

Figure 27: Unauthorized user tries to connect to database

5.2.2 Authorization for REST API

The authorization for the REST API is thoroughly described in Section 4.5.1. The authorization has been comprehensively tested during the development part. In Figure 24, an unauthorized user tries to log in, which gives the status code 401, meaning that the user was not given access. Unit test was also made to test the authorization part of the REST API.

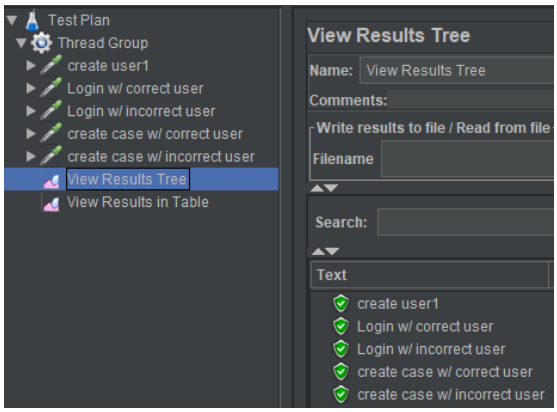

Figure 28: Unit test for REST API

Figure 28 shows the unit test made for the authorization part of the REST API. Assertions were made to check if the requests that was supposed to fail did. As shown in Figure 28, they all passed, which points to that the authorization works.

5.2.3 Risk assessment for ISO 27001

Risk	Probability	Consequence	Risk value	Fix
Access to db	2	2	4	Change passwords
Access to AWS	2	2	4	Generate new private key
Password leakage	3	3	9	Change passwords
Access to source code	3	3	9	New safety method

Figure 29: Risk assessment for the whole system

Figure 29 shows the risk assessment made for the project. The probability and consequence ranges from 1 to 5, where 1 is very unlikely, and 5 is very likely. The probability are made from how likely we think this sort of problem is to happen, as the developers of the system. The consequence is made out of how much of a problem we consider the risk to be. The risk value is the product of the two. If the risk value goes over 12, it is a problem that needs to be fixed straight away. Although the scenarios is rather unlikely to happen, it is still possible. DH Anticounterfeit is a small company, and the likelihood of a attack against them is quite small. The problems are all fixable, however, if the source code leaks, it is quite hard to get it back, or remove if it is put up on the internet. That is why it is important to keep it safe from the beginning.

6 Discussion & conclusion

Since a lot of the decisions regarding what software components to use was made for us, we did not have to research which components would work best for us, which was a time-saver. Most of the time was spent designing and developing the REST API, since that contains most components, and the fact that everything has to work together, makes it the most complex part. All tests made with JMeter could be done more thoroughly. This could in return lead to some undiscovered errors which could then me repaired. Now that the project is coming to an end, answers to the questions stated in the beginning can be given.

6.1 Answers to main questions

How to design a REST API with the ISO 27001 standard in mind?

- As described in Section 4.5.3, the ISO 27001 standard contains a lot more than software related security. DH Anticounterfeit is currently certified for this specific standard, and they recommended us to do a risk assessment, which seems to be the most common thing when discussing ISO 27001. The risk assessment was made with the loss of data, and unauthorized access in mind.
- Other than the risk assessment, a lot of focus was put into the security, and authorization, which is described in Section 4.5.1. Since there are so many different kinds of attacks, and ways to manipulate a system, security was considered a big and important part of the project.

What modifications on the current app have to be made to work with the REST API?

- Since the old app works in a certain way around the email-sending function, a lot has to be changed. Every time that an API call has to be made, an HTTP request has to be sent. This can be done using curl commands, which was used to test the API and has to be created in C# code. This has to be done every time a case is created, changed or deleted. Every time a user logs in, or out, and every time a picture is saved. Since no user registration existed initially, this had to be done as a modification to the mobile application.
- The old app is quite poorly optimized, especially if a back-end is used with it. In order to get the app to work properly, a complete redo would be appropriate. This way the app could be built around the REST API, which would be more efficient.

The third question is about how and why a back-end system is superior to the old method that was used. Of course, this all depends on the system. However, the advantages are substantial when the 2 versions are compared.
What makes a REST API better than the current method?

- When data is being sent over email, it has to be handled manually. This costs both time and money, and is not very efficient. With a back-end system, everything is handled automatically, and you are able to handle the data in a way the you could not when it it transferred over mail.
- Using a proper back-end system could also lead to a performance boost. Instead of saving data locally, as the application did, everything is stored on the database, freeing up resources for other functions.

6.2 Social requirements

- **Integrity** is a part of this project since it is handling personal data, and takes pictures. However, it does not include sensitive personal data, and the application is mostly used in Asia, which dodges the laws of The General Data Protection Regulation (GDPR).
- **Rules & safety:** During the project no laws or rules had to be taken into consideration. The application was given to us, and the data that is being handled was already defined. The developer of the mobile application looked into the rules regarding what data can be handled, for example accidental pictures of other human beings. Because of this, we did not spend time looking into this.
- **Environment** aspect of the project is rather hard to determine, since it is pure software. However, what the application might lead to can affect the environment in a positive way. Counterfeit products can help support terrorist organizations, and fund even more terrorist attacks. If more customers would use a well-functioning application to fight counterfeit products, less attacks could happen.
- **Economy:** Because of the use of a domain supplied by DH Anticounterfeit, a free version of AWS and a free SSL-certificate supplied by https://letsencrypt.org, there has been no financial cost for the project. However, a lot of time was spent on the project.. If a similar project would be done at a company, the cost would be higher, since no personnel costs are included in our estimation.

6.3 Future improvements

There are always improvements to be made in every system. One of the biggest things that would need to be done is proper testing of the entire system. As described earlier, due to lack of time, testing of the whole system has not been done. Overall the API requests responds as expected to the tests that we have put them through under different conditions. Another improvement would be the mobile application. To develop the application around the API, and even use native development, would increase the user experience a lot, and a faster and more bug-free application could be achieved.

6.4 Conclusion

The application now communicates with a completely new back-end system built from scratch. The system complies with all the requirements that we put on the system: REST API has to be used, the API follows the OPEN API standard, Docker is used, and it uses MongoDB as a database. If more time was available, proper testing of the system would be possible. The plan was to do this, but the end of the project was spent writing on the thesis. The structure and planning of the system could have been better. Things never go as smoothly as planned, and some parts of the project took longer than anticipated. When looking at the whole project, we are happy with the system and the result. If we planned the project better, the same result could have been achieved in shorter time.

References

Online resources

[1] *Dh anticounterfeit*, "Home Page" https://www.dhanticounterfeit.com/, Accessed: 2019-02-18.

[2] *Xamarin app development with visual studio*, https://visualstudio.microsoft.com/xamarin/, Accessed: 2019-02-18.

[3] *Instagram*, "Instagration Pt. 2: Scaling our infrastructure to multiple data centers" https://instagram-engineering.com/, Accessed: 2019-03-06.

[9] *Firebase*, https://firebase.google.com, Accessed: 2019-05-15.

[10] *What is http?* https://www.w3schools.com/whatis/whatis_http.asp, Accessed: 2019-03-14.

[12] *Amazon s3*, https://aws.amazon.com/s3/?nc=sn&loc=1, Accessed: 2019-04-29.

[13] *Amazon elastic container service*, https://aws.amazon.com/ecs/, Accessed: 2019-05-06.

[14] *Amazon ec2*, https://aws.amazon.com/ec2/, Accessed: 2019-05-06.

[17] *Graphql*, https://facebook.github.io/graphql/draft/, Accessed: 2019-03-14.

[20] *MySQL*, "Home Page." https://www.mysql.com, Accessed: 2019-05-16.

[22] *What is a container?* https://www.docker.com/resources/what-container, Accessed: 2019-03-14.

[26] *What is python? executive summary*, https://www.python.org/doc/essays/blurb/, Accessed: 2019-03-14.

[27] *Apple*, "About Swift" https://docs.swift.org/swift-book/, Accessed: 2019-03-05.

[28] *Oracle*, "Java™ Programming Language" https://docs.oracle.com, Accessed: 2019-03-04.

[29] *Microsoft*, "Introduction to the C# Language and the .NET Framework" https://docs.microsoft.com/en-us/dotnet/csharp/, Accessed: 2019-03-05.

[30] S. Hidayatullah, *Man in the middle attack prevention strategies*, https://www.computerweekly.com/tip/Man-in-the-middle-attack-prevention-strategies, Accessed: 2019-03-14.

[32] *Amazon ec2 instance types*, https://aws.amazon.com/ec2/instance-types/, Accessed: 2019-05-06.

[33] *Amazon compute service level agreement*, https://aws.amazon.com/compute/sla/, Accessed: 2019-05-06.

[34] *Python restful web framework generator*, https://github.com/guokr/swagger-py-codegen, Accessed: 2019-02-18.

[35] *Nginx wiki*, https://www.nginx.com/resources/wiki/, Accessed: 2019-05-06.

[36] *Let's encrypt root trusted*, https://letsencrypt.org/2018/08/06/trusted-by-all-major-root-programs.html, Accessed: 2019-05-06.

Thesis

[5] R. Aniceto at el., "Evaluating the cassandra nosql database approach for genomic data persistency," *International Journal of Genomics*, 2015, Accessed: 2019-03-07. [Online]. Available: http://dx.doi.org/10.1155/2015/502795.

[8] I. Costa, J. Araujo, J. Dantas, E. Campos, F. Airton Silva, and P. Maciel, "Availability evaluation and sensitivity analysis of a mobile backend-as-a-service platform," 2015, Accessed: 2019-03-11. [Online]. Available: https://doi.org/10.1002/qre.1927.

[15] R. H. Canaday, R. D. Harrison, E. L. Ivie, J. L. Ryder, and L. A. Wehr, "A back-end computer for data base management," *Communications of the ACM*, Accessed: 2019-03-10. DOI: 10.1145/355620.361172.

[18] A. Vazquez-Ingelmo, J. Cruz-Benito, and F. J. García-Penalvo, "Improving the oeeu's data-driven technological ecosystem's interoperability with graphql," 2017, Accessed: 2019-03-14. DOI: 10.1145/3144826.3145437.

[19] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, Accessed: 2019-03-07. DOI: 10.1145/1735688.1735706.

[21] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2013, Accessed: 2019-03-07. DOI: 10.1109/PACRIM.2013.6625441.

[23] C. Anderson, "Docker [software engineering]," *IEEE Software*, 2015, Accessed: 2019-03-08. DOI: 10.1109/MS.2015.62.

[24] P. E N, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of docker containers based on hardware utilization," *International Conference on Control Communication & Computing India (ICCC)*, 2015, Accessed: 2019-03-11. DOI: 10.1109/ICCC.2015.7432984.

[25] A. Charland and B. LeRoux, "Mobile application development: Web vs. native," *Queue - Data*, Accessed: 2019-02-28. DOI: 10.1145/1966989.1968203.

[31] G. Disterer, "Iso/iec 27000, 27001 and 27002 for information security management," *Journal of Information Security*, 2013, Accessed: 2019-03-11. DOI: 10.4236/jis.2013.42011.

Books

[4] K. Douglas and S. Douglas, *PostgreSQL: A Comprehensive Guide to Building, Programming, and administering PostgreSQL databases*. Sema2005, 2005.

[7] M. S. Jan Stryjak, *The Mobile Economy 2019*. GSM Association, 2019.

[16] S. Jan and M. Sivakumaran, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media; 1 edition, 2011.

Marcus Grimberg

Ludwig Nord

PO Box 823, SE-301 18 Halmstad
Phone: +35 46 16 71 00
E-mail: registrator@hh.se
www.hh.se