Introduction

Last edited by **Freja Marie Madsen** 1 year ago

Data can be intercepted while transferring or breached at storage locations. Data can be used for malicious purposes, like blackmailing people, accessing accounts, selling data or likewise. The aforementioned happenings can cause real harm to real people. Services, like Google, Facebook and more, that store sensitive and non-sensitive but exploitable data must therefore protect obtained data and render it unusable in case of security breaches, through methods of encryption. Furthermore, unsecure databases can easily be breached, edited, destroyed, or held for ransom if not secured and backed up properly, and may cause great property damage to companies. Therefore, Data Security is an important part of software engineering and something that is important for developers to understand and implement in their programs.

The overall project consists of twelve teams working on a music streaming service, which is intended to be available for free and expected to be used by a large number of concurrent users. The final product will be a streaming service, which can stream several musical products. This project will be designed based on microservice architecture, which means that the music streaming service will be built as independent components that function as a service for the overall application. Each service communicates through API's, and each performs a single function.

As a module of this product, our microservice must ensure that the user's password can be collected and stored securely regarding the requirements from privacy laws such as GDPR. This means passwords and other sensitive data must be encrypted, preferably with one way encryption. Our microservice is therefore responsible for storing passwords and other sensitive data if present. In case of a breach, this data must be unreadable, to those who intercept it. Should server failure or similar data loss occur, the data must be stored elsewhere to ensure that at least one version of data is always available. It is essential that the system is tested and drilled to ensure that the system delivers the required functionality, capacity, and safety.

Our module communicates with other microservices through an API which functionally works as a CRUD interface. This interface offers a method of manipulating a secure database through a set of operations consisting of create, get, post, and delete. Other microservices will be able to use our service to store a password and to check if a password exists for a given user.

Analysis and requirements

Last edited by **Klaes Drescher Sørensen** 1 year ago

The purpose of this segment is to obtain a thorough and detailed understanding of data security in order to further understand and determine requirements needed for our part of the music streaming service program. Requirements will function as guidelines throughout the project and help ensure that the program achieves desired functionality/performance.

GDPR

The application will be able to store data concerning users, data that will enable users to use the services. However, when working with personal data there are some privacy laws which must be followed. These laws dictate what is allowed to be saved, and how it is to be saved.

Personal data is any type of information that can be redirected to a specific person, even if the person can only be identified if the information is combined with other information. There are different kinds of personal information, non-sensitive data or sensitive data. Non-sensitive data can be identifying information such as name, address, age and education. It can also be applied to financial matters, other purely private matters or service matters. It can also be information in the form of pictures, fingerprints, exam results and so on. Sensitive data is any data that is about racial or ethnic origin, political opinion, sexual orientation and religious or philosophical beliefs. However, it is only in extremely rare cases that sensitive personal data is processed. The module will only be concerned with non-sensitive data.

Regarding the storage of personal data, article 32 in the General Data Protection Regulation (GDPR) requires that the controller and the processor shall implement appropriate technical and organizational measures to ensure a level of security appropriate to the risk. A way of doing this if for example to encrypt personal data as it states in article 32 1.a. [6]

Furthermore, it states in article 17 in the General Data Protection Regulation (GDPR) also known as “the right to be forgotten”, that subjects have the right to request that any information about them is deleted. [5]

Any personal information that will be stored such as passwords will be stored in a safe manner and protected with e.g. encryption. The users must have the ability to delete their account and also delete all their personal data from our database.

Cap theorem

In order to choose which type of database management system that will fit the program the best, it is essential to look at each database system and what that system supports. This is important because as stated by the CAP theorem, in case of a network failure a distributed system can only deliver two of three desired characteristics at once, which are consistency (C), availability (A) and partition tolerance (P).

Database management systems (DBMS) that are in the AP category will allow high availability and partitioning tolerance in the event of network failure. This means all nodes are available, but not all are updated. If a user connects to a bad node, outdated data is received. This could cause problems during sign in, as some users may have recently changed account details, loading incorrect data to the user. When the network failure is resolved the nodes will synchronize to ensure eventual consistency. AP DBMS’es are e.g. Cassandra, CouchDB and MongoDB.

DBMs in the CP category allow for consistency and partition tolerance in the event of a network failure. When a network failure occurs the system turns off inconsistent nodes until it is resolved. CP systems are structured so there is only one primary node that receives all the requests in a replica set. The secondary nodes then replicate the primary node and work as a backup in case the primary node fails. CP DBMS’es are e.g.Redis and MongoDB.

Lastly CA databases are able to provide consistency and availability across all nodes. Unfortunately however they can not deliver fault tolerance meaning it can not function properly in the event of a network failure. In distributed systems network failures are bound to happen which makes CA databases impractical. CA DBMS’es are e.g. MySQL and Postgres. Both AP and CP databases could fit our application, but since the highest priority is availability, and temporary inconsistent data can be tolerated, an AP database system is the best option for this project. [2]

As AP database systems are Non relational databases (NoSQL-based) they have the advantage of having no limits on what types of data to store. This means if it is later on decided to store new types of data, it will not encounter issues storing it. AP databases are designed to be fragmented across multiple shards, which makes them very scalable and excellent at handling “big data”, since there is no need to enforce strict data structures. Therefore this type of database is better than a CA database at storing larger amounts of data, as they scale horizontally, whereas CA databases scale vertically. Horizontal scaling is also more cost efficient. [11]

When it comes to NoSQL databases there are four different types. Graph stores, Column stores, Key-value stores and Document stores. For this project Cassandra, which is a Column store type, has been chosen as the database system. [15]

Since most of the information in the database is stored in columns, aggregation queries are pretty fast, which is important if the project requires a large amount of queries. The system needs to be able to support millions of concurrent users. Column-store databases are very good for this, as they can expand nearly infinitely using horizontal scaling. [13]

Programming language

Python is an open-source development tool. This means everyone can contribute to the programming language with libraries and likewise. Python has a reputation for being user friendly as it has libraries for almost anything and thereby offers a huge variety of tools and solutions to common problems that programmers may run into. This makes python a practical, powerful and flexible tool for developing applications.

These tools can be imported into any projects by using the PIP (Preferred Installer Program). The group already knows of several powerful tools for python, for every task that the project requires. These tools include: Flask, Unit Test, Bcrypt, psycopg2, RESTful and more. [14]

Communication between microservices

As part of this project a microservice architecture has been used, it is a designed way to build a larger project using modular services. The modular services handle different components of the larger project, this could be some groups handling front end, back end, database integrations, login or other features. The different microservices would then be connected using application programming interface, API, this would be the communication connection between the different microservices ensuring that the different modules are able to connect to one another and create a larger project. Using a microservice paradigm provides the development teams with a decentralized way to build each part of the application. This creates a way for each service to be isolated, rebuilt and redeployed in an independent way. This is also great if a microservice fails it is easy to locate which service is failing and then testing that service and refactoring it. Using microservices brings the usage of API's, which is a sort of contract between two services, when service one sends a specifically structured request then they can expect a specifically structured return from the second service. This can help simplify an entire process of implementing new components to an application because it is known what other components already do and how the existing architecture is developed. [10]

There are a couple of ways to establish communication between microservices. The most used is via RESTful Web API end-points. But there is also gRPC integration, Messaging integration and Streaming integration. RESTful Web API end-points and gRPC integration are two very similar ways to revive the goal of communication between microservices. These two methods are good in scenarios where the user initiating the request must wait for a response. They are both low latency because they do not require other components. However gRPC integration does mostly provide the lowest latency because of this the Protobuf protocol used.

Securing user data

When you want to hide data such as passwords you can encrypt them. There are many ways of doing this but they all try to turn the password from plain text into an encrypted text. However, this may not be enough to secure the passwords. This is because encryption is a two-way function. Therefore it is reversible. One can always decrypt the encrypted string and get the original one if they figure out how the text/password has been encrypted. A way to circumvent the use of a two-way function is by using the method of hashing. Hashing is a one-way function which means that it only can be decrypted if the algorithm is known or by guessing the original text and comparing it to the hash. Hashing works by taking a data array as an array of an arbitrary length and converting it into a fixed length by using a hashing table. Using hashing there will also not be stored any password of users directly in a database, instead the hash of the password is stored. When it needs to be checked if the password exists in the database, the entered password of the user is then hashed again and the current hash from the user and the stored hash is compared, the user would then be logged on or denied access based on the outcome. [8]

Hashing algorithm

There exists a plethora of different hashing algorithms, with different capabilities, some of these are MD-5, RIPEMD-160, Whirlpool and SHA. Algorithms such as SHA, come in different versions creating hashes of different bit sizes like SHA-256 or SHA-512. These forms of algorithms are described as fast, they are notorious for sacrificing some security in exchange for low compute time. They are therefore more vulnerable to brute force attacks and rainbow table/look table attacks, which are tables with already cracked hashes. A way to make these fast hash algorithms more secure is with the addition of salt. Salt is the addition of an extra string added to the password, they are then both hashed making the password longer. The salt is a variable, being randomly generated for each hash and making sure no hash uses the same salt making the brute force attack more complicated, time consuming and hardware consuming by increasing the entropy of a hash. [4]

Key stretching

As computers evolve and follow Moore's Law, where the idea is that transistors on microchips double in power every two years even though the price of the computers is halved. [12]

This makes for a great threat to hashing algorithms even with added salt that is not updated to ensure that the time to crack hashes is longer, by not using a way to keep the hashing algorithm scalable with Moore's Law the hashed passwords become more susceptible to being cracked as time goes on. It is therefore important to keep updating hashing algorithms in terms of security to keep up with this evolving technology. Using key stretching is a way to mitigate into a form of slow hashing, using iteration upon the hashed string and salt. This can be scaled to the same rate as the evolving technology. A great key stretching algorithm would be Bcrypt that is based on the blowfish cipher, that is already created to take a string input, a salt and has an adaptive function for iteration essentially making it slower. The downside of using key stretching algorithms is making the whole process slower and annoying to the users and creating extra work load on the server. [7]

SQL injection

While it is important to save the user data in a secure manner in the database, the database itself must also undergo a safe protocol as to not be in danger of an SQL injection attack (SQLi) and necessary precautions must be taken to ensure this. SQLi attacks against insecure databases will allow attackers to modify and collect data of the database. They will be able to run SQL commands and be able to add, delete, alter and find data. Such an attack is a huge threat to the data of the users and can cause a major breach of the entire database. To secure the system from SQLi attacks it is important that the code is written properly and that the database is never directly interacted with. The code should include prepared statements, and the database should only be interacted with through input contained in these prepared statements. In addition, the code should be sanitized and written according to common practices. Users of the system must never be able to see SQL errors as it could help an attacker discover more information about how the database is built and how it works. [1]

Birthday attacks

A birthday attack is a form of attack on hashed data derived from brute force attacks, where the attacks implement the mathematical phenomenon of the birthday paradox, where the probability of 23 people in the same room there is at least a 50% chance that two people share the same birthday. The way attackers implement this is that they use probabilistic logic to minimize the complexity of obtaining a hash collision and determining the estimated risk of the presence of a hash mismatch within a given number. Therefore, finding a particular hash collision is more challenging than finding a matched hash collision with the same values. The basic way to defend against such an attack is by increasing the entropy of hashes, this could be achieved by using key stretching that is described under the part "Key stretching". [9]

Backups and restoring data

Another attack that can harm a database is a ransomware attack. Ransomware attacks are a type of attack where a hacker acquires a database, encrypts contained data so it is unreadable. A method of protecting against ransomware attacks is to save backups both offline and off-site. An offline offsite backup, physically disconnecting it from remote access, can prevent irreversible effects of a ransomware attack. Though it can be complicated and time-consuming to restore data from an offline location and online offsite backups such as cloud backups/online backups generally are the easiest to restore data from. it is generally better to have both types of backups in case the database is attacked.

When making and storing backups you can use the 3-2-1 method. This method advises you to have at least three copies of your data, two local (on-site) but on different media/locations and at least one copy off-site. This method ensures that the system does not have a single point of failure.

It is also important to make new backups often and regularly. How often a database needs to be backed up, depends on how often the data changes. If the database has over 2000 updates per hour, and it is only backed up twice per day, a relatively large amount of data will be lost if errors occur.

It is strongly recommended to test the data recovery process using available backups. This will certify that backups are created correctly and that it will be able to successfully restore data from the backup. Tests will also help make an incident response plan that will make sure that an organization/company has a specific plan to restore data in the event of e.g. a ransomware attack. [3]

Requirement Specification

This segment contains the requirement specification for the application. It describes what the software will do through the functional requirements(FR - figure 1) and how it will be expected to perform through the non-functional requirements (NFR - figure 2).

Each requirement is prioritized using the MoSCoW method which is a popular prioritization technique for managing requirements. It represents four categories of initiatives: must-have (M), should-have (S), could-have (C), won't-have (W), or will not have right now (R).

Functional requirements:				
ID	**Description**	**Analysis**	**Validation section**	**MoSCoW**
FR-1	To store data safely at all times.			
FR-1-1	There must be a backup available, where the data is consistent.	Backups and restoring data	Database	M
FR-1-1-1	Backups should be done regularly, at least once every 24 hours.	Backups and restoring data	Database	S
FR-1-1-2	Backups should be saved offline to protect against ransomware threats.	Backups and restoring data	Database	S
FR-1-1-2-1	Backups could also be saved online in order to make it easier/faster to restore the data.	Backups and restoring data	Database	C
FR-2	The database should be protected against the most common attacks.			
FR-2-1	The database must be protected against brute force attacks.	Securing user data	Brute force attack - time test	M
FR-2-1-1	The data should be salted to prevent the use of rainbow tables in brute force attack.	Securing user data	Brute force attack - time test	S
FR-2-1-2	The password should be hashed with an SHA algorithm, in this case bcrypt.	Securing user data	Brute force attack - time test	M
FR-2-1-3	The password should be key-stretched after being hashed a given amount of times to prevent brute force attack.	Securing user data	Brute force attack - time test	S
FR-2-2	The database should be protected against birthday attacks.	Securing user data		
FR-2-2-1	The database should be salted with variable salt to prevent birthday attacks.	Securing user data	Brute force attack with rainbow table - time test	S
FR-2-3	The database should be protected against SQL injections.	Securing user data	SQL injection attack	
FR-2-3-1	The query should use prepared statements when calling the database with user related input requests.	Securing user data	SQL injection attack	M
FR-2-3-2	Our program should use prepared statements.	Securing user data	SQL injection attack	M
FR-2-3-3	The program user should only have the required database privileges.	Securing user data	Check privileges	C
FR-3	Our program must have API-endpoints that allow other microservices to communicate with ours.			
FR-3-1	The program must be able to respond to requests made by other microservices.	Communication between microservices	Request tests	M
FR-3-2	The api json body requests should be validated.	Communication between microservices	Request tests	M
FR-4	The user should be able to delete his or her userdata.	What data do we store and how do we store it	Database - query (CRUD)	S

Figure: 1 - Functional requirements table.

NON-Functional requirements:				
ID	**Description**	**Analysis**	**Validation section**	**MoSCoW**
NFR-1	Our program should be able to perform as expected when at least 1.000.000 users are using the system		Request tests	M
NFR-2	Our api should be written in python	Programming language		M
NFR-3	The api should store data in a postgreSQL database.	Cap theorem (what is important for our system - what can support this)		M
NFR-4	The api should be able to respond to all users' interactions (sign in, sign up, delete) after maximum 2 seconds	Cap theorem (what is important for our system - what can support this)	Request tests	S
NFR-5	Our way of storing user data must meet the laws and requirements for personal data and the GDPR rules	What data do we store and how do we store it		M
NFR-6	Our API documentation should describe how to use our endpoints.			M
NFR-6-1	Describe what parameters are required.			M
NFR-6-2	Describe what parameters are optional (if any).			M
NFR-6-3	Describe what response is expected for a given request.			M
NFR-7	Have an understanding of which groups are using our services.	Securing the whole system (IP)		M
NFR-7-1	Give the other groups proper resources on how to use our service.	Securing the whole system (IP)		S

Figure: 2 - NON-Functional requirements table.

References

1. Acunetix. (s.d.). What is SQL Injection (SQLi) and How to Prevent It. Acunetix.com. Lokaliseret den 10. 09 2022 på https://www.acunetix.com/websitesecurity/sql-injection/
2. Choosing Database: CAP Theorem. (s.d.). Nexsoftsys. Lokaliseret den 23. 12 2022 på https://www.nexsoftsys.com/articles/CAP-theorem-database-dbms.html
3. Congionti, Victor. (2020, 11. 06). Keeping Your Backups Safe from Ransomware Attacks. InfoSecurity. Lokaliseret den 10. 09 2022 på https://www.infosecurity-magazine.com/opinions/keeping-backups-ransomware/
4. Defuse Security. (2021, 28. 09). Salted Password Hashing - Doing it Right. CrackStation.net. Lokaliseret den 09. 10 2022 på https://crackstation.net/hashing-security.htm#salt
5. General Data Protection Regulation (GDPR): Art. 17 GDPR Right to erasure ('right to be forgotten'). (2020). I: GDPR.EU. Lokaliseret den 23. 12 2022 på https://gdpr.eu/article-17-right-to-be-forgotten/
6. General Data Protection Regulation (GDPR): Art. 32 GDPR Security of processing. (2020). I: GDPR.EU. Lokaliseret den 23. 12 2022 på https://gdpr.eu/article-32-security-of-processing/
7. John D., Cook. (2019, 25. 01). Salting and stretching a password. JohnDCook.com. Lokaliseret den 09. 10 2022 på https://www.johndcook.com/blog/2019/01/25/salt-and-stretching/
8. Jscrambler. (2016, 22. 10). How to Store Passwords Safely. Blog.JScrambler.com. Lokaliseret den 10. 09 2022 på https://blog.jscrambler.com/how-to-store-passwords-safely/

9. Kathuria, Sonal. (s.d.). What is Birthday Attack? How Can You Prevent Birthday Attacks?. SecurityPilgrim.com. Lokaliseret den 09. 10 2022 på https://securitypilgrim.com/what-is-birthday-attack-how-can-you-prevent-it/

10. Mathews, Sasha. (2022, 25. 01). 4 Ways to Establish Communication between Microservices: With the main pros and cons of each. LevelUp.GitConnected.com. Lokaliseret den 10. 09 2022 på https://levelup.gitconnected.com/4-ways-to-establish-communication-between-microservices-984207f29497

11. Schaffer, Erin. (2021, 04. 11). System design fundamentals: What is the CAP theorem?. Educative.io. Lokaliseret den 10. 09 2022 på https://www.educative.io/blog/what-is-cap-theorem

12. The Editors of Encyclopaedia Britannica. (2022, 02. 09). Moore's law: computer science. Britannica.com. Lokaliseret den 09. 10 2022 på https://www.britannica.com/technology/Moores-law

13. Tobin, Donal. (2021, 25. 11). Which Modern Database Is Right for Your Use Case?. Integrate.io. Lokaliseret den 10. 09 2022 på https://www.integrate.io/blog/which-database/

14. W3Schools. (s.d.). Python PIP. I: W3Schools.com. Lokaliseret den 10. 09 2022 på https://www.w3schools.com/python/python_pip.asp

15. Williams, Alex. (2021, 12. 02). Column-Oriented Databases, Explained. KDNuggets.com. Lokaliseret den 10. 09 2022 på https://www.kdnuggets.com/2021/02/understanding-nosql-database-types-column-oriented-databases.html

Design and development

Last edited by **Rasmus Helleberg Eriksen** 1 year ago

After having established the requirements for the system, a system design was made which was used to guide the implementation of the system. The purpose of this segment is to show the envisioned software solution and how it was implemented. Visually designing systems before implementation, allows developers to confront problems preemptively, saving time and resources by avoiding refactoring. It segments a common understanding of the final product, how it will work and how it is to be implemented.

Flow Chart:

In order to plan the process and design for the api, a flowchart is a good tool to help identify the essential steps from a higher perspective. A flowchart can give a good easy understanding of what is expected of the usage of the Api's endpoints. As shown in figure 3 we know when to return what depending on the request being received.

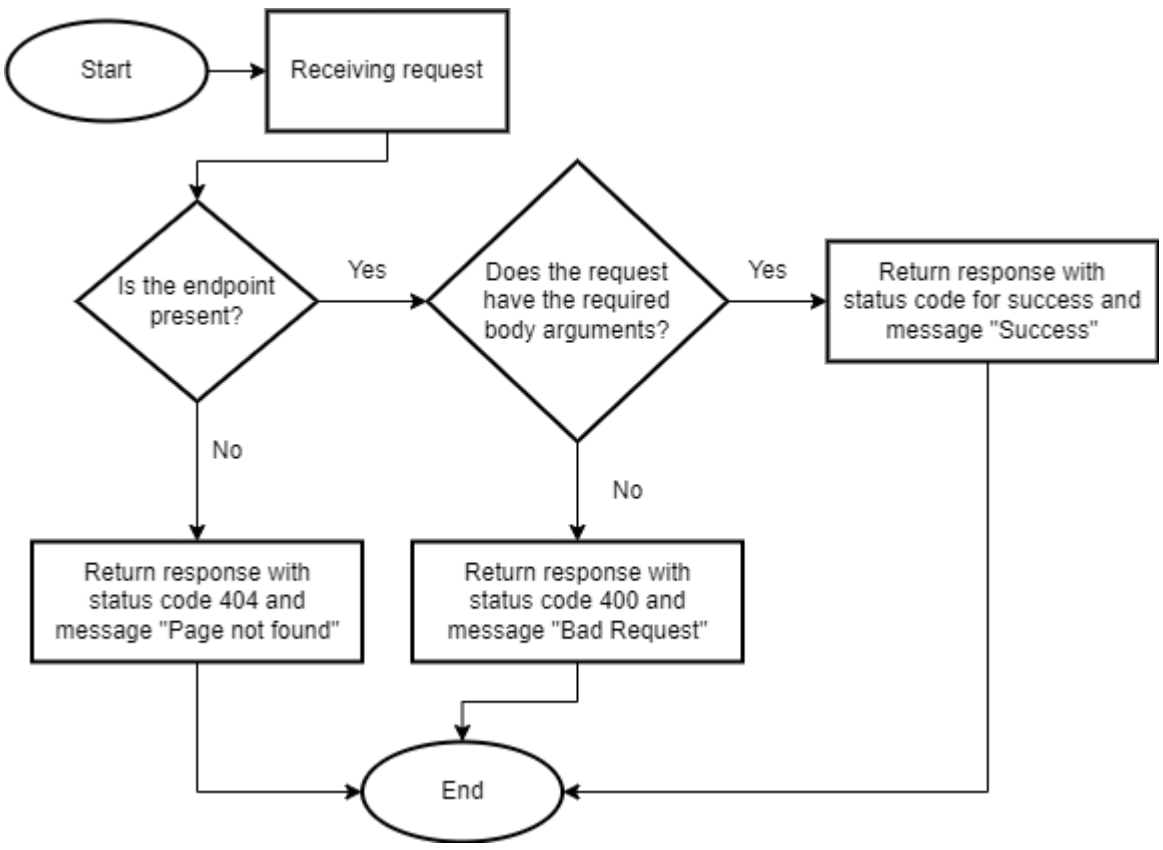

Figure: 3 - API Flow Chart.

C&C schema:

Through analysis the conclusion found was that it was best to make both local and cloud backups. Based on the analysis it was also decided that it was needed to do two local backups and one cloud backup to be as secure as possible so that there will always be an accessible backup. The following C&C Shema, figure 4, shows how components should be set up and how they should communicate. It should be possible to make and save backups from the database both locally and in the cloud.

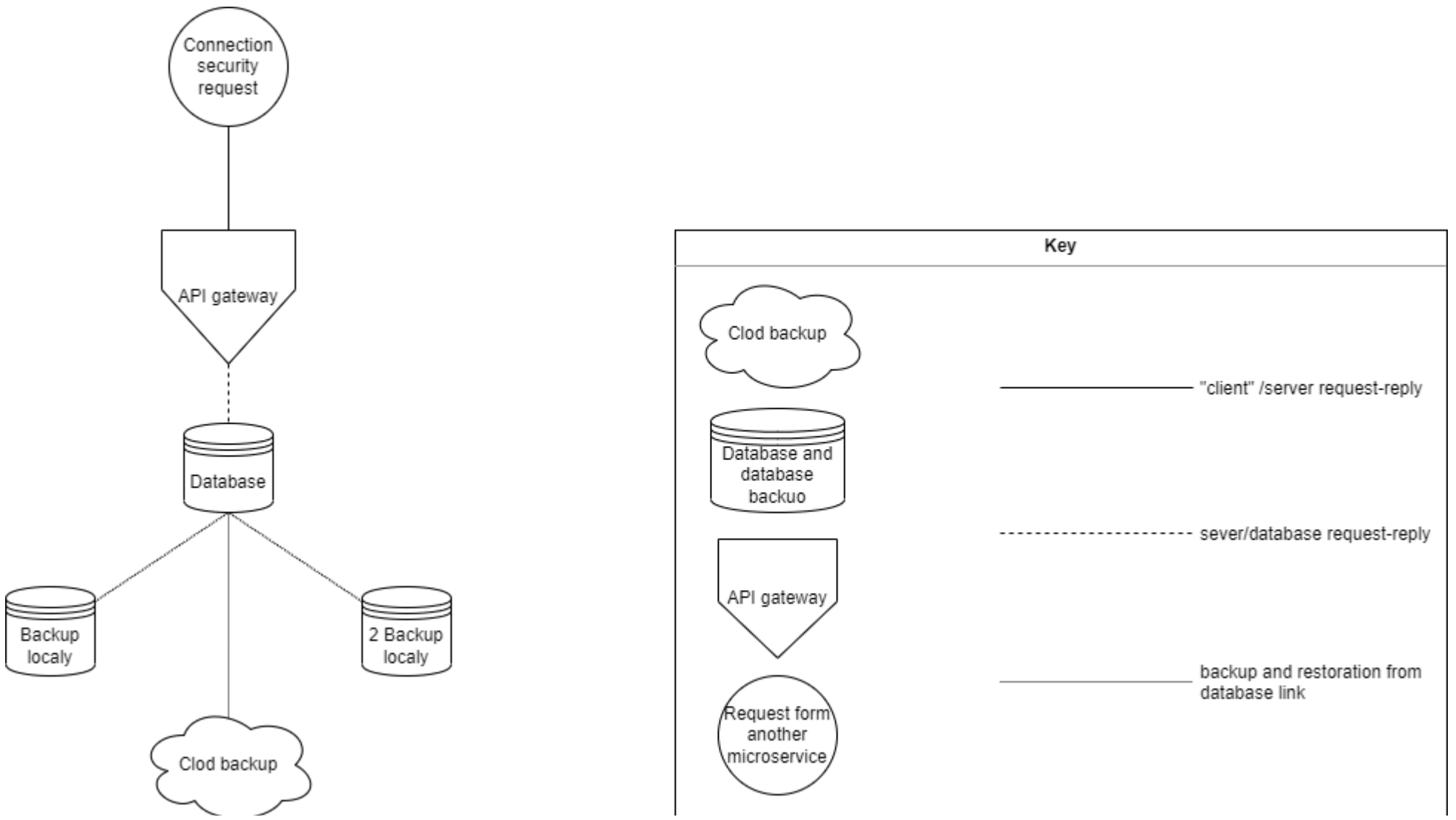

Figure: 4 - Microservice C&C diagram.

Activity Diagrams:

Activity diagrams visually present a series of actions or flow of control in a system. It is a useful tool for finding constraints and conditions that cause a particular event. In order to determine the events of the log in and the register method two activity diagrams were made. The two methods were chosen because they make up the most important functionality for the microservice.

The flow of activities, when a user with a profile tries to log in.

The system should get a login request which tells the system to take the sent user data, salt it, hash it, and then check if the hashed data exists in the database. If it does, the login attempt will be successful, and a success message would be sent back. If it is not found, then an error message will be sent back.

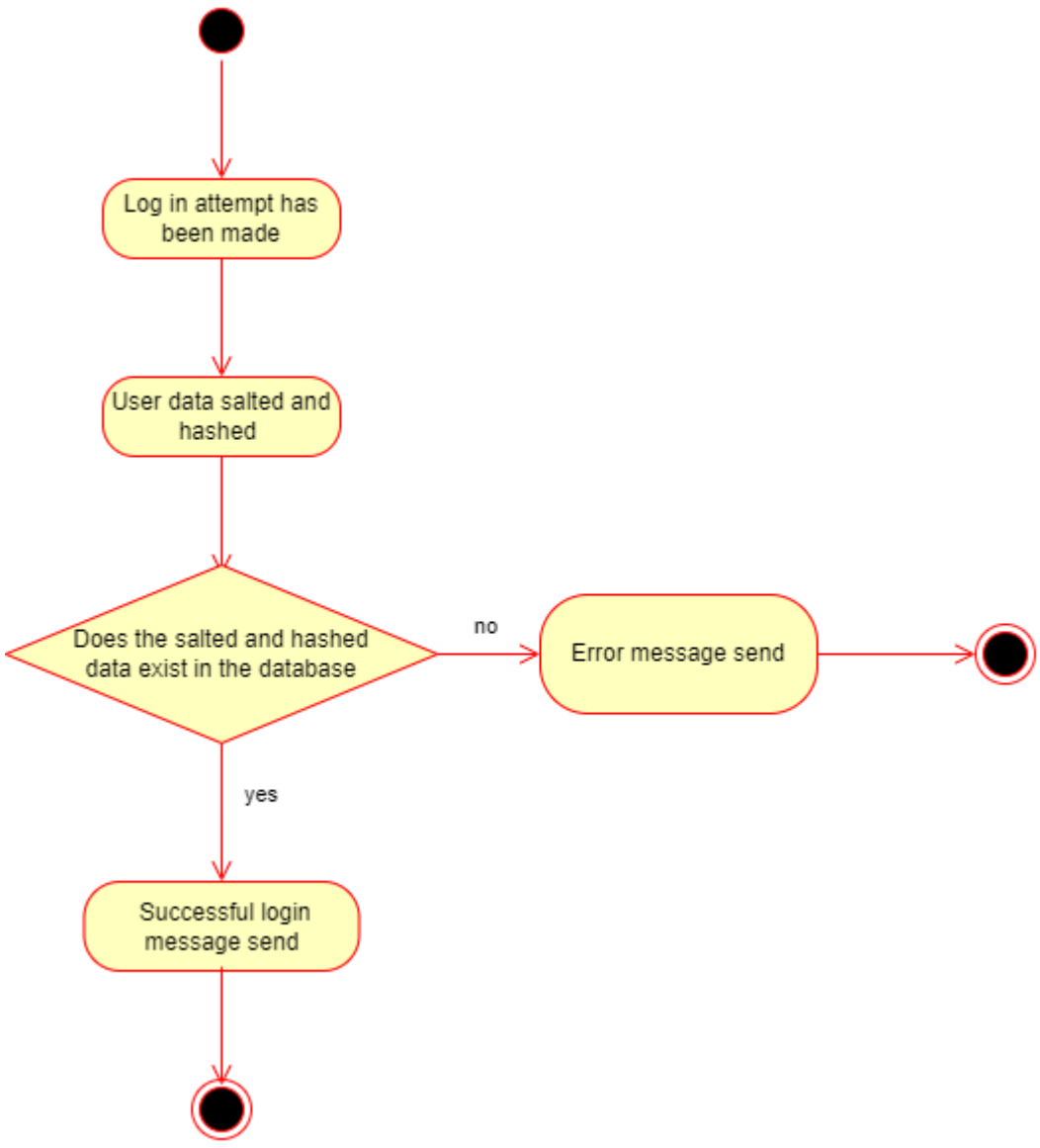

Figure: 5 - Activity diagram for user login.

The flow of activities, when a new person tries to register themselves as a user.

The system should get a register request which tells the system to get the new user data, salt it, hash it, and then check if the hashed data exists in the database. If it does, an error message will be sent back. If it is not, then a new user with the sent user data is created, and this user is then stored in the database.

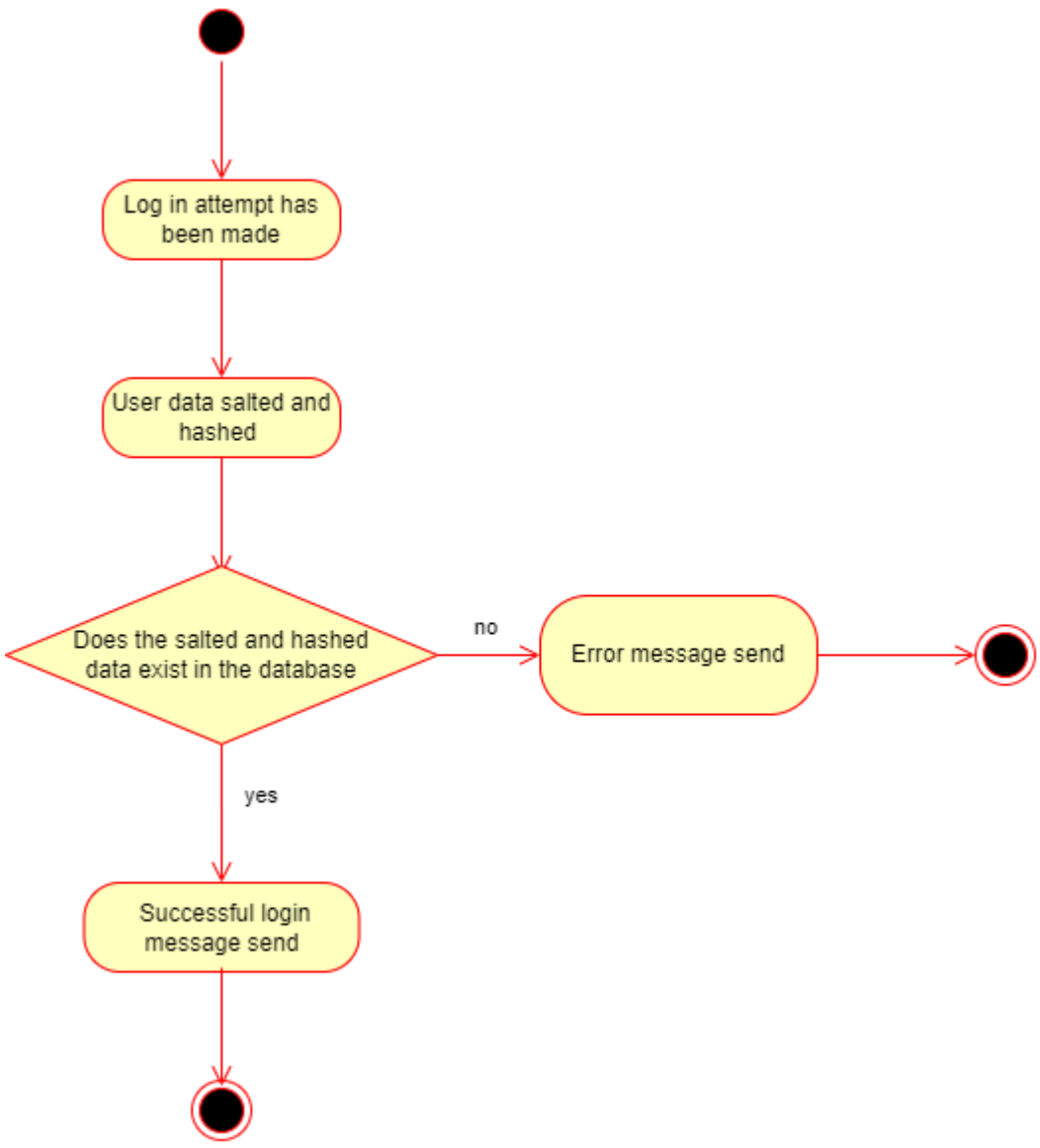

Figure: 6 - Activity diagram for user registration.

Block Diagram:

A block diagram is a visual representation of a system that uses simple, labeled blocks that represent single or multiple items, entities or concepts, connected by lines to show relationships between them. In order to make an overview of our microservice and its concepts. As shown in the figure 7 when a request is made to the whole system Kubernetes then sends a request to our microservice if one of the other microservices makes a request upon it. Then the load balancer receives the request and picks the server with the least amount of work to fulfill the request. The server can then connect to the database if needed and return the needed information to the microservice that made the request to ours.

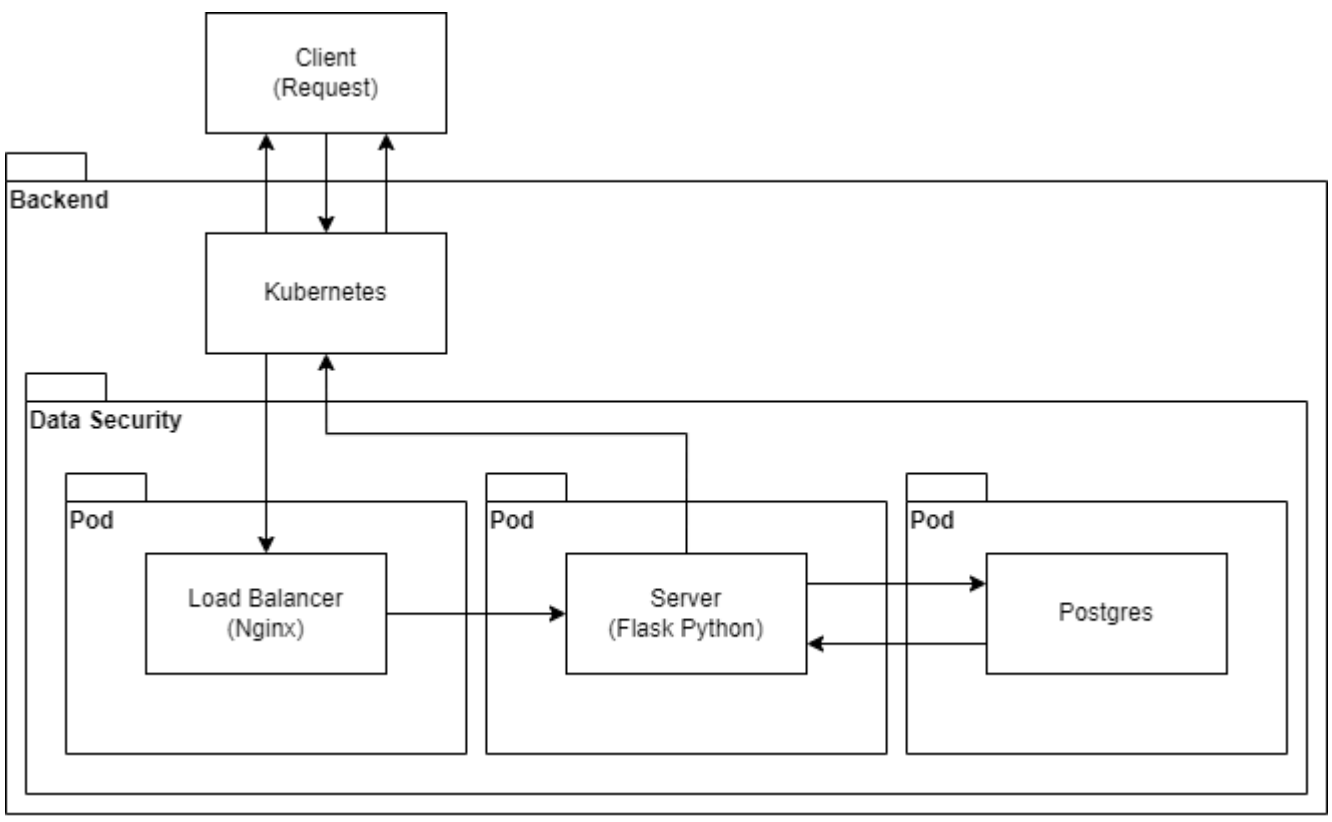

Figure: 7 - Block diagram of Data Security microservice on Kubernetes cluster.

Database Design:

Database design diagrams divide information into subject-based tables and determine the relations between tables in order to reduce redundant data and ensure accuracy. The larger and more complex a database becomes, the more the utility of a detailed and accurate database design increases. The Data Security microservice does not have a large or complicated database; as it only has to store a single table. A database diagram was created regardless to ensure every member had a common idea as to how the database was to be implemented, and as documentation.

user_table	
PK	id INTEGER UNIQUE NOT NULL
	user_email VARCHAR(255) UNIQUE NOT NULL
	user_password VARCHAR(255) NOT NULL
	user_salt VARCHAR(255) UNIQUE NOT NULL

Figure: 8 - Initial database table for users.

Sequence diagram:

A sequence diagram is a form of UML diagram that illustrates the sequence of messages between objects during an interaction. The timeline for the interaction begins at the top and descends gradually as the sequence of messages comes to an end.

In the activity diagram, it is shown how the messages flow from one activity to another. Here the use of a sequence diagram deepens the understanding of the flow of the system, by showing how the messages flow from one object to another object. We have chosen to do sequence diagrams for the same functionalities as the activity diagrams. This also means that it will not include the hashing and salt.

User exists request

In this sequence diagram it is shown how a request upon a user's existence on the database is carried out. There will be a call with user specific data, then the database will be checked for that user. In case the user exists a success message is sent back, if the user does not exist an error message will be sent back.

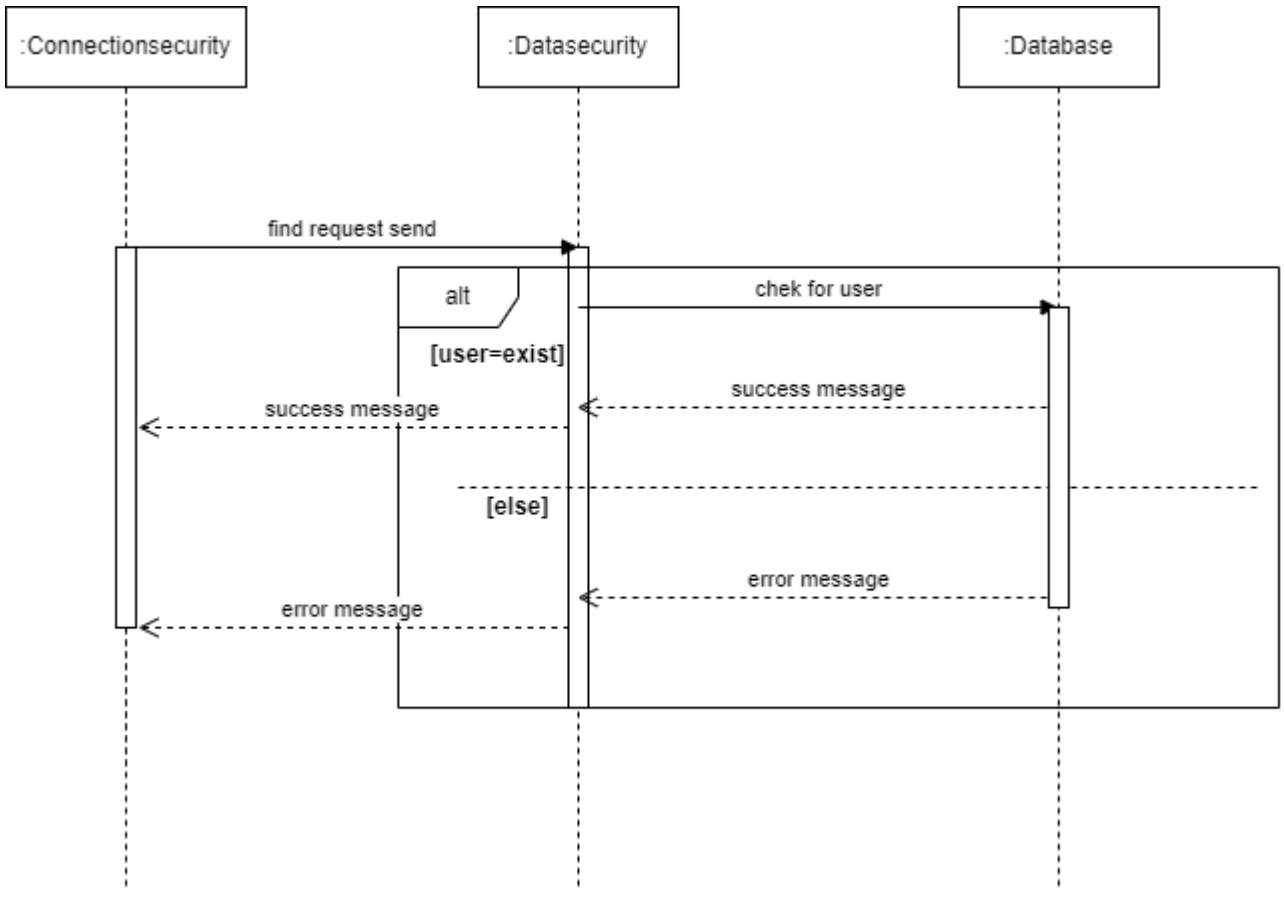

Figure: 9 - Sequence diagram for checking if a user exists.

Login attempt

The second diagram shows the process of a login request. The attempt to login will come with a user’s data, then it will be checked if the user does exist and if the data is correct in the database. If the data is correct a success message will be sent back or in case the user does not exist an error message will be sent back.

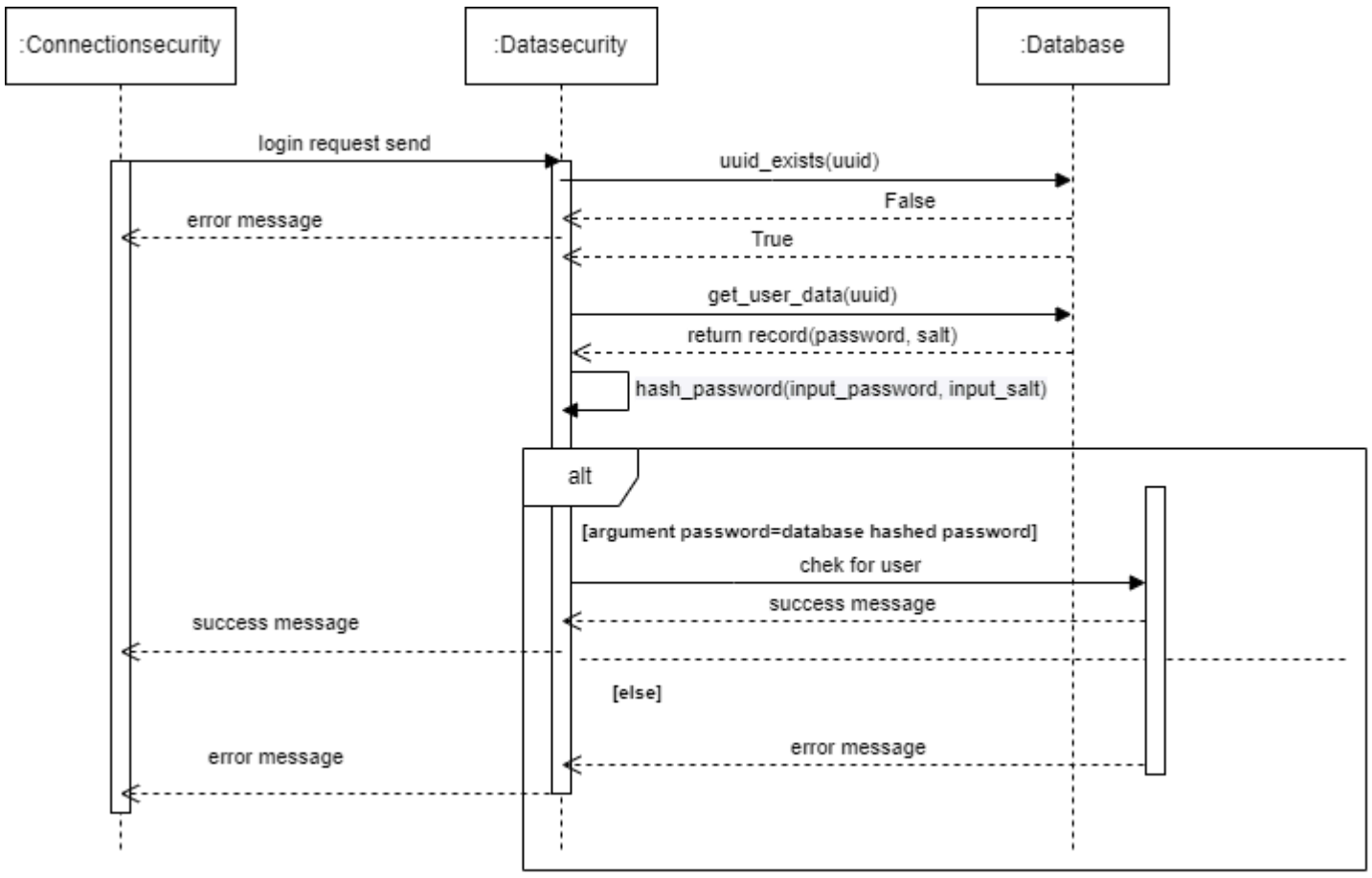

Figure: 10 - Sequence diagram for user login attempt.

Registration

In the third diagram the process of registering a user is shown. The process will come with data being sent on a new user, the system will check if an already existing user is in the database, in that case a message of the existing user will be sent back. If the user does not exist the user will be stored in the database and a success message will be sent back.

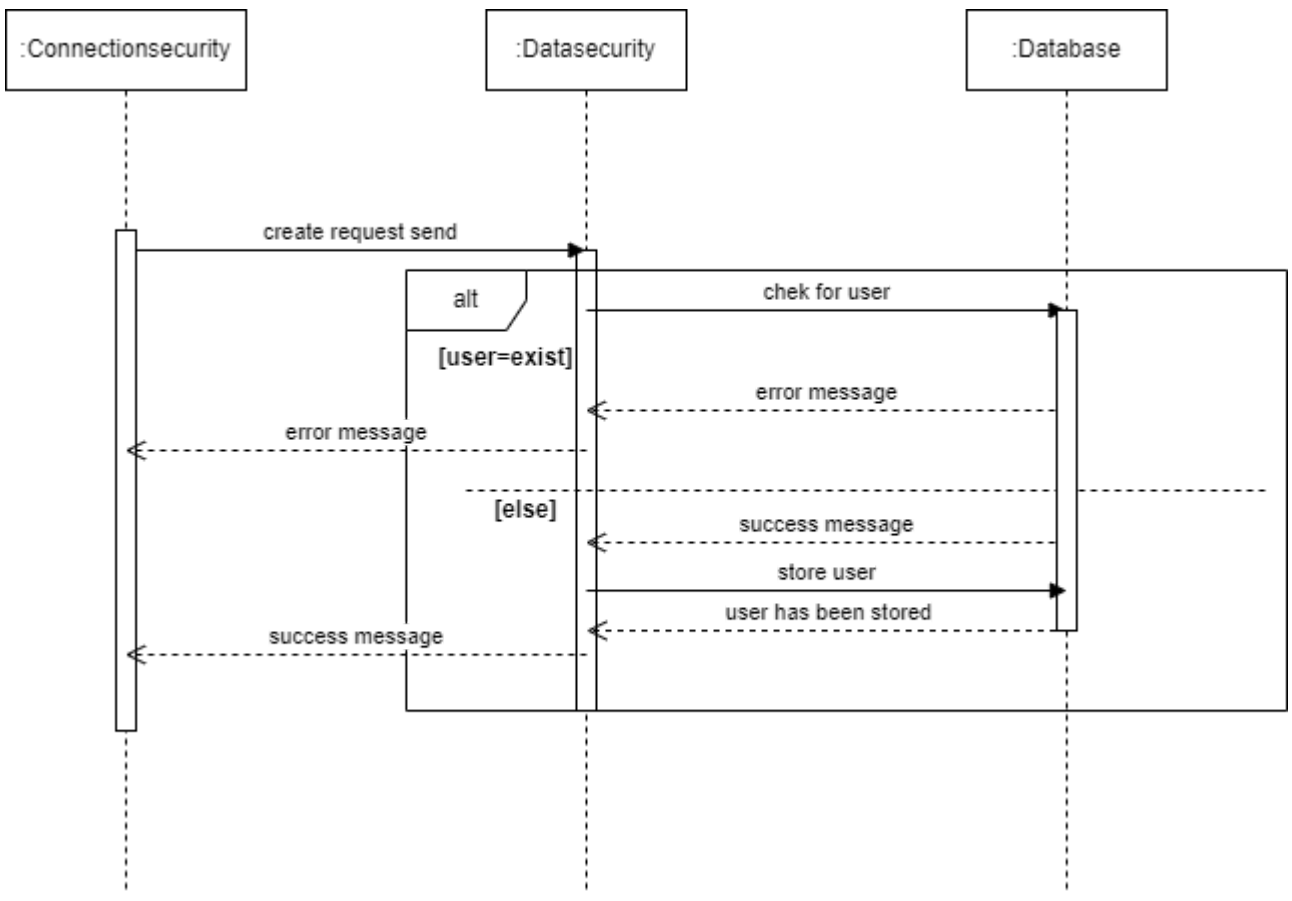

Figure: 11 - Sequence diagram for user registration.

Limitations and constraints

During the project certain complications arose and the implementation of certain aspects were not as intended. So in reality this setup became too advanced for the scope of the project. This meant that we did not have time to achieve this. If there was more time this would be an important and good addition to the application.

The database language

Through the analysis it was clear that a NoSQL database language such as Cassandra would be a better option for our application because it fitted the abilities of a CP system. However, there were a lot of problems with the implementation of this type of database. The setup was extremely complicated and time-consuming. After having tried to implement this type of database for some time it was decided that there would not be enough time to finish the application with this type of database. It was then decided that a PostgreSQL database should be used instead. As mentioned in the analysis this type of database is a CA system and not the most optimal solution. Postgres is still a valid choice and because of the simplicity of our requirements it does not conflict with the desired functionality. It was far more practical for the group to work with postgreSQL, as every group member was already familiar with the tool and it would help us save a significant amount of time and create space for other aspects of the project.

Backups and restoration

There were difficulties regarding the backups, as substantial configuration with Kubernetes had to be completed. At the time of implementing backups, the project was approaching the deadline. While we had enough time to implement backups, we could no longer reasonably demand group 12, Operations, to help us with this implementation, meaning backup within the cluster was no longer feasible. We did not predict this problem would arise, and therefore addressed it too late. Therefore we're not able to follow the C&C diagram and the 3-2-1 method.

Architecture

The Data Security microservice communicates only with the Connection Security micro service. When a user attempts to login to an account, the Connection Security microservice forwards a password, plaintext or hashed, and a UUID fetched from the Subscription Service microservice using an email to identify which UUID is connected to the user attempting to login. The Data Security service then hashes the password with a salt fetched using the given UUID, which is then matched with the encrypted password stored at the location of the user with the passed UUID. If the password matches, an "200 ok" response is then returned through the API. This UUID is originally generated when the Connection Security micro service attempts to create a new user in the database. This UUID is returned to Connection Security micro service, to Connection Security micro service, which will then forward it to relevant microservices, like the Subscription Service micro service, to use as foreign keys for userdata to be connected to. Creating a user takes only a password. If no errors occur before reaching the Data Security micro service, a new user is created, a salt and UUID generated, and the password hashed. All generated items are then stored in the database as a user. Finally the UUID, along with a "201 created" response, is returned through the API. Other operations, like, "update password" and "delete account", require both a correct password and UUID to perform successfully, returning a relevant response through the API. The final operation manipulating the database is "reset password", which requires only a UUID. This function should be used only as a last resort if users have no access to their account and should preferably be used in tandem with external methods of security, like email confirmation or likewise.

Implementation

Bcrypt

One of the main requirements for the project was that the database should be protected against the most common attacks, such as brute force attacks, rainbow table attacks and birthday attacks. This meant that a hashing algorithm matching criterias established in the analysis, had to be found. Bcrypt is a hashing algorithm library that, in this module, takes an input value and adds a 16-byte random salt-value to it and makes a 24-byte hash. Bcrypt also uses key-stretching which slows down the algorithm. Bcrypt was therefore chosen as our hashing algorithm and used to make sure that our data were stored securely.

```python
# Method that generates a salt
def generate_salt():
    return bcrypt.gensalt()

# Method that takes the password_input and a salt
# that has been generated for that user and hashes it
def hash_password(input_password, input_salt):
    # hash with salt
    return bcrypt.hashpw(input_password.encode('utf-8'), input_salt).decode('utf-8')
```

Figure: 12 - Code snippet from our hashing.py file.

Regarding the security of the data it was also important that the data were protected from SQL injections. This was solved by using prepared statements for our queries. This ensures valid and safe inputs and prevents any tampering with the data or the database.

```python
try:
    cursor = connection.cursor()

    create_query = "INSERT INTO user_table(user_uuid, user_password, user_salt) VALUES(uuid_generate_v4(), %s, %s) RETURNING user_uuid"
    cursor.execute(create_query, (password, salt))
    connection.commit()

    record = cursor.fetchone()
```

Figure: 13 - Code snippet from our db_conn.py file.

However as stated in the analysis it's only possible to store the salt in the database as plaintext. Although this could come with the possibility of a brute force attack since the data is readable, it ensures the hashes entropy is kept to a higher degree and prevents rainbow/look up table attacks. The best way to prevent brute-forcing is simply to use long, complex passwords.

Database

It was initially assumed that the Data Security microservice was solely responsible for user authentication. The database and program was at first designed and implemented with this in mind, meaning the database would store a user email and a password for authentication. It was later discovered that the Subscription Service microservice intended to store email meaning the email would be saved in more than one database, resulting in duplicate data. It was then determined that the Data Security microservice should no longer save the email to avoid duplicate data, meaning that the login process was split across two microservices. At it's core, this does not conflict with the Data Security microservice requirements, and was therefore not a matter that needed further discussion. Following this decision, relatively significant refactoring was needed to both the database and program, removing data and functions regarding storage and manipulation of user emails. During this process, the Data Security microservice was also asked to provide a UUID rather than simply an id. This was a minor change, demanding very little extra work.

user_table	
PK	id INTEGER UNIQUE NOT NULL
	user_uuid VARCHAR(255) UNIQUE NOT NULL
	user_password VARCHAR(255) NOT NULL
	user_salt VARCHAR(255) UNIQUE NOT NULL

Figure: 14 - Final user table in database.

Api

To get an idea of how the API should work, flowcharts were used(figure 3). This made it easier to implement all features the API needed, as they were determined pre-implementation.

To implement the microservice api, the Flask framework, with the Flask-RESTful extension was used. This made it possible to create endpoints that other microservices in the Kubernetes cluster could make requests to. First the body arguments for a specific endpoint were defined. As shown by figure 15, a RequestParser instance is created, where the argument “password”, defined as a string, is required, and a message “Password is required” is returned in case of missing arguments.

```
# Request Parser for user - This defines what we expect the request to contain (* register args *)
user_args_register = reqparse.RequestParser()
user_args_register.add_argument("password", type=str, help="Password is required", required=True)
```

Figure: 15 - Code snippet from our main.py file.

Logic for the specific endpoint was then created, as shown by figure 16. The endpoint is defined as a POST request that requires arguments as defined in figure 15. If arguments are satisfactory, the endpoint post method is successfully executed. To help implement this endpoint and the endpoint for the log-in procedure, sequence diagrams (figure 9, figure 10 and figure 11) and activity diagrams (figure 5 and figure 6) were used.

```
# Check if the uuid exist before registering a new user
class Register(Resource):
    def post(self):
        args = user_args_register.parse_args()

        # If the uuid does not exist then a new identifier will be made
        try:
            salt = hashing.generate_salt()  # Make salt

            #created_uuid = hashing.generate_uuid(salt)  # Make uuid
            hashed_password = hashing.hash_password(args['password'], salt)  # Hash the password-input
            uuid = db.create_user_identifier(hashed_password, salt.decode('utf-8'))
            return { "uuid": uuid, "status": 201, "message": "User was created."}  # Return the unique user id for the identifier
        except:
            return {"status": 500, "message": "Something went wrong..."}  # User already exists
```

Figure: 16 - Code snippet from our main.py file.

At last, the endpoint was added to the api resources so that the web server knows where to point the request given on a specific url, shown by figure 17.

```
# Add resources to the api and create the endpoints
api.add_resource(Register, "/api/v1/register")
```

Figure: 17 - Code snippet from our main.py file.

When all endpoints were created, a documentation was created to help other microservices understand and utilize the Data Security micro service, API Documentation.

Kubernetes

When the code for the api was finished, deployment of the Kubernetes cluster was needed for the other microservices to utilize the Data Security microservice. The first step was to create a docker image of the api shown in figure 18. After building the image, it was pushed to the gitlab repository where it could be pulled for deployment to the Kubernetes cluster.

```
# start by pulling the python image
FROM python:3.8

# copy the requirements file into the image
COPY ./requirements.txt /app/requirements.txt

# switch working directory
WORKDIR /app

# install the dependencies and packages in the requirements file
RUN pip install -r requirements.txt

# copy every content from the local file to the image
COPY . /app

# Postgres connect enviroment variables
ENV POSTGRES_SERVICE=postgres-service
ENV POSTGRES_PORT=5432

# configure the container to run in an executed manner
EXPOSE 5000

CMD [ "python", "main.py" ]
```

Figure: 18 - Code snippet from our dockerfile.

Then the Kubernetes config yaml files were made figure 19. The config file `api-deployment.yaml` creates a deployment of the api where it pulls the docker image from the Data Security microservice gitlab container registry and specifies the container port. Then the config file `api-service.yaml` creates a service for the api deployment where it is specified where ingress must point to, using the ClusterIP. Then other microservices can use the url `prod-team10.c3.themikkel.dk` + the specific endpoint they wish to call. Example: `prod-team10.c3.themikkel.dk/api/v1/register`.

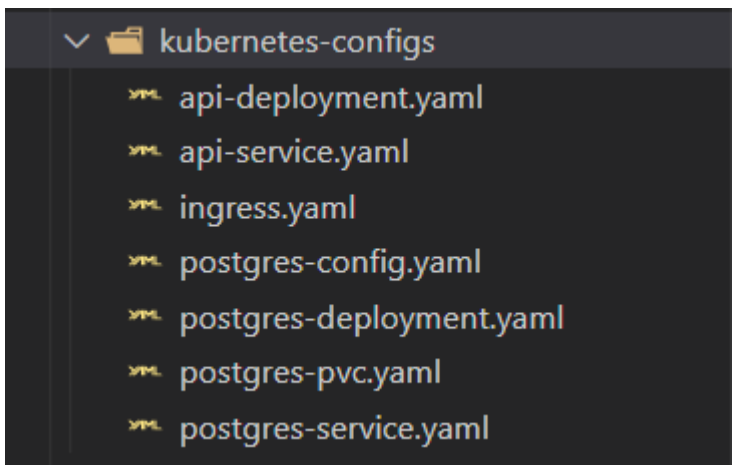

Figure: 19 - Kubernetes config files structure.

Backup of database

The idea of creating a backup for the database was to ensure that the ability to restore lost data in case the database fails, is tampered with or Kubernetes is malfunctioning in any way that could harm the database. Creating backups of the database would ensure that it is possible to restore lost data that might have been lost due to a failure or malfunction. As previously mentioned in the analysis, it is also important to do multiple backups as the 3-2-1 method suggests.

How we should do backups

Creating this backup file of the database would be done periodically, with the usage of a cron job. A CronJob is a linux command that is used for periodically scheduling a task that is to be run, this task would be run in a specific time period in which would be calling on a script that would run a task that takes a backup of the database and stores it. The backup of the database would be stored both in a cloud-based system such as AWS or Google Cloud Storage and on a file that is stored on the Kubernetes server. The script that would be called using the cron job would be through the service of PgAdmin that is used for the SQL framework, PgAdmin uses the `pg_dump` utility command to create a plain-text or archived format for an easy way to restore the backup using SQL commands. This way of making backups would enable us to follow the 3-2-1 method and the C&C diagram.

What we were able to do

Local backups was made from a local database manually by using the following command from the directory where the database data is stored:
`./pg_dump.exe -U postgres -d my_database_name -f D:\Backup\<backup-file-name>.sql`

The command is based on the postgres `pg_dump` utility. To execute this command, opening Powershell, going to the Postgres bin folder and then executing the mentioned command to dump the data would be required. If CronJob was used for making backups, it would have been exactly the same command, as used locally. It would just have been executed by the CronJob instead.

Restoration of data from backups

As mentioned in the analysis it is also strongly recommended that the recovery process is regularly tested. It is important that the restoration happens as fast as possible in order to reduce the down time. The way to ensure this is by making an incident response plan and regularly attempting to restore data from the backups to ensure that it works. As mentioned in Limitations and Constraints there was not enough time to set up a database with Kubernetes.

How we should do restoration

Had it been possible to make a backup with `pg_dump`, stored on the Kubernetes server, it would have been possible to manually restore the database using this command on the kubernetes cluster: `kubectl exec -i [pod-name] -- psql -U [postgres-user] -d [database-name] -f <backup-file-name>.sql`

What we were able to do

We were, however, able to manually make a local backup from a local database, which we also were able to restore data from, with this command: `/psql.exe -U [postgres-user] -d [database-name] -f D:\Backup\<backup-file-name>.sql`

The command is based on the Postgres “psql” utility which is used to restore text files created by pg_dump. To execute this command, open Powershell, go to the Postgres bin folder and execute the mentioned command. This command is similar to the command which should have been used if CronJob was implemented.

General conclusion

Last edited by **Rasmus Helleberg Eriksen** 1 year ago

In this project, the group was tasked with securing sensitive data relating to the music streaming service, this entails encryption of passwords, understanding what the intellectual property of the project was and how to protect it, understanding what GDPR is and how it could affect the project, and creating a backup system for the database. Finally all features must be tested to prove claimed functionality.

Throughout the project, the main focus was setting up a database for user data subjected to the rules of GDPR regarding sensitive data. Storage of passwords has been accomplished in accordance with today's encryption standards, through the usage of Bcrypt. Endpoints have been created in order for other microservices to utilize the Data Security microservice. The service successfully runs on the project kubernetes cluster, and is reachable from other microservices.

A domain model was made in order to convey the final implementation for the application. Since the program is not based on class relations but is based on endpoints calling functionality, the domain model has been changed with regards to the domain model conventions. Every call to an endpoint returns a singular response, meaning there is a 1-1 relation between caller and endpoint. All endpoints return a message of validation to the caller. The only exception is register, when registrering the caller will receive both a UUID of the newly registered user and a validation response. Every call from an endpoint to the database will result in a return of validation.

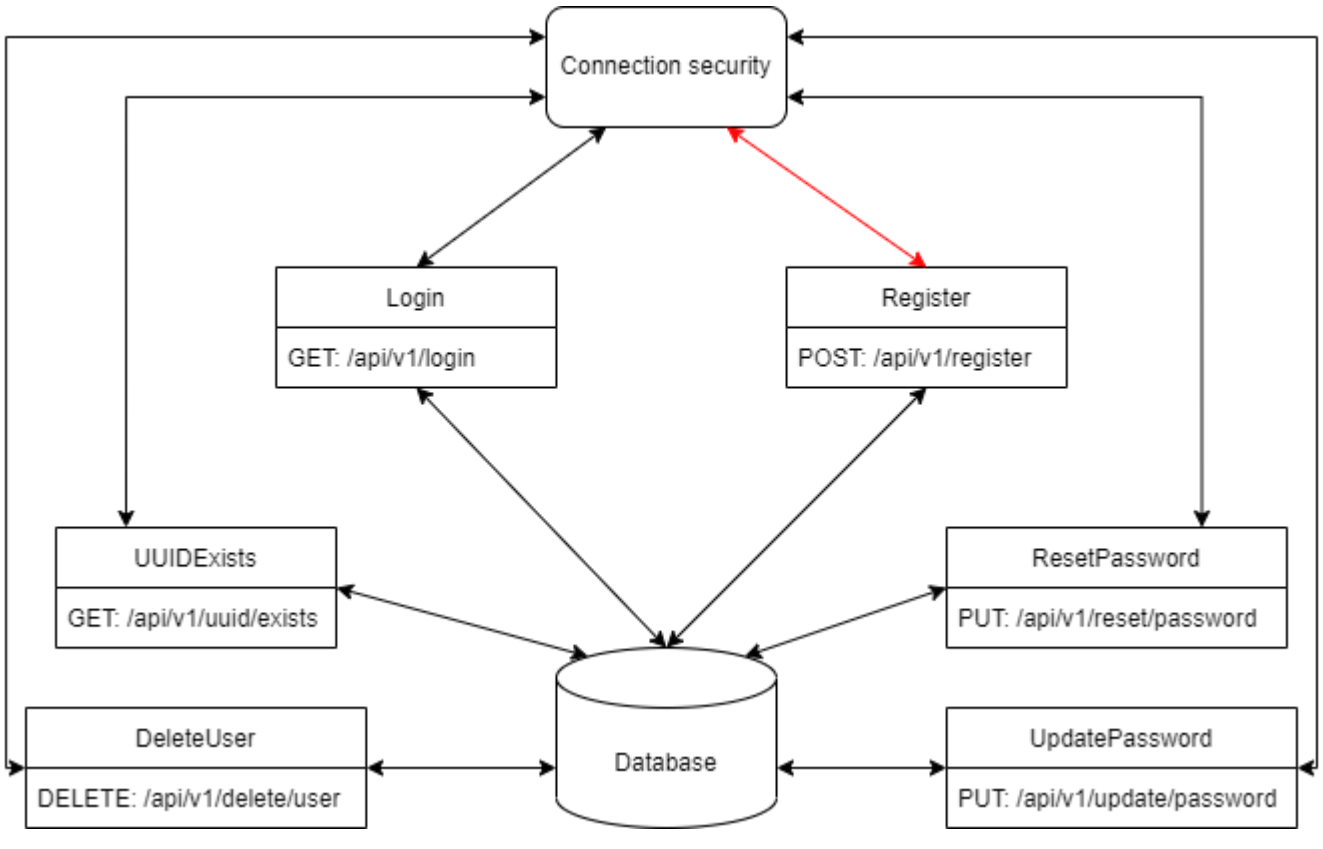

Figure: 27 - Api domain model

In conclusion, although most functionalities exist and have been proved to work, a complete data security microservice, as described by the requirements, has not been implemented. Existing functionalities can be used regardless of the absence of remaining features, but may lack full security and capacity.

API Documentation

Last edited by **Rasmus Helleberg Eriksen** 1 year ago

Table of contents

Register

Use case Create a uuid and password, returns the uuid.

URL : `/api/v1/register/`

Method : `POST`

▼ **More Information - Click here**

Data constraints

Requires a json body:

```
{
    "password": "[In plain text - String]"
}
```

Data example

```
{
    "password": "examplePassw0rd"
}
```

Success Response

Condition - 1 : User was created.

```
{
    "uuid": [
        "2fcfcfd9-e89a-4965-ac50-b62cc3136556"
    ],
    "status": 201
}
```

Error Response

Condition - 1 : Something went wrong...

```
{
    "status": 500
}
```

Login

Use case Checks if the argument password matches the password that correspond to the argument uuid.

URL : `/api/v1/login/`

Method : `GET`

▼ **More Information - Click here**

Data constraints

Requires a json body:

```
{
    "uuid": "[128-bit label to identify the user]",
    "password": "[password in plain text - String]"
}
```

Data example

```
{
    "uuid": "f939e182-0095-425c-b947-f8573e317cda",
    "password": "examplePassw0rd"
}
```

Success Response

Condition - 1 : UUID exists - Login successful.

```
{
    "status": 200
}
```

Error Response

Condition - 1 : User credentials incorrect.

```
{
    "status": 404
}
```

UUID exists

Use case Check if a user with a give UUID exists.

URL : `/api/v1/uuid/exists`

Method : `GET`

▼ **More Information - Click here**

Data constraints

Requires a json body:

```
{
    "uuid": "[128-bit label to identify the user]"
}
```

Data example

```
{
    "uuid": "f939e182-0095-425c-b947-f8573e317cda"
```

```
}
```

Success Response

Condition - 1 : UUID exists.

```
{
    "status": 200
}
```

Error Response

Condition - 1 : UUID does not exist.

```
{
    "status": 404
}
```

Update password

Use case Update password using old password.

URL : `/api/v1/update/password`

Method : `PUT`

▼ **More Information - Click here**

Data constraints

Requires a json body:

```
{
    "uuid": "[128-bit label to identify the user]",
    "new_password": "[In plain text - String]",
    "old_password": "[In plain text - String]"
}
```

Data example

```
{
    "uuid": "da754db2-6f9b-4c59-95c7-fd30459f83e9",
    "new_password": "newPassw0rd",
    "old_password": "oldPassw0rd"
}
```

Success Response

Condition - 1 : Password updated succesfully.

```
{
    "status": 204
}
```

Error Response

Condition - 1 : User credentials not updated.

```
{
    "status": 304
}
```

Condition - 2 : User credentials incorrect.

```
{
    "status": 404
}
```

Reset password

Use case Reset password using email. (Only allow access if password was forgotten)

URL : `/api/v1/reset/password`

Method : `PUT`

▼ **More Information - Click here**

Data constraints

Requires a json body:

```
{
    "uuid": "[128-bit label to identify the user]",
    "password": "[In plain text - String]"
}
```

Data example

```
{
    "uuid": "da754db2-6f9b-4c59-95c7-fd30459f83e9",
    "password": "newPassw0rd"
}
```

Success Response

Condition - 1 : Password updated succesfully.

```
{
    "status": 204
}
```

Error Response

Condition - 1 : User credentials not updated.

```
{
    "status": 304
}
```

Condition - 2 : User credentials incorrect.

```
{
    "status": 404
}
```

Delete user

Use case Delete user.

URL : `/api/v1/delete/user`

Method : `DELETE`

▼ More Information - Click here

Data constraints

Requires a json body:

```
{
    "uuid": "[128-bit label to identify the user]",
    "password": "[In plain text - String]"
}
```

Data example

```
{
    "uuid": "da754db2-6f9b-4c59-95c7-fd30459f83e9",
    "password": "passw0rd"
}
```

Success Response

Condition - 1 : User deleted succesfully.

```
{
    "status": 204
}
```

Error Response

Condition - 1 : User not deleted.

```
{
    "status": 304
}
```

Condition - 2 : User credentials incorrect.

```
{
    "status": 404
}
```

Status Codes

Data Security returns the following status codes in its API:

Status Code	Description
200	`OK`
201	`CREATED`
204	`RESCOURCE UPDATED`
304	`NOT MODIFIED`
400	`BAD REQUEST`
404	`NOT FOUND`
500	`INTERNAL SERVER ERROR`