DLS project report 2023

Rasmus Kristian Koefoed, William Jean Kristensen Omø,
Samavia Sophia Salim & Nikolaj Bruun-Hansen

SD22/SD23v
Date: 26th May 2023

Repositories, deployed services and guides
can be found at the bottom of this report

Table of contents

1. Introduction

In this project, we were tasked with creating a large, distributed system. The goal was to showcase an understanding of creating scalable system architecture and the ability to integrate microservices, use authentication and authorization, and interoperability.
Our banking application contains admin management, customer management, account management, and transaction processing functionalities.
We have divided the codebase into two components, which are the admin backend and the customer backend.
The admin backend handles the administrators' management, while the customer backend handles the management of customers in the application.
We aimed to ensure better organisation and maintainability of the codebase by separating the admin service from the customer service.
Our project lacks some of the more sophisticated patterns and a better logging system but otherwise meets the requirements for the project.
Throughout the report, we will describe our system in further detail and highlight the challenges encountered during the project.

1.1. Problem description

Which architecture and why is best suited for the structure of a large, distributed system? Which patterns should be deployed, and what is their purpose? What are the pros and cons of hosting the various services, and what does this mean for the overall architecture? What changes can be made to make the project more scalable? What strategies should be used by the team to increase efficiency?

1.2. Requirements

1.2.1. Functional Requirements

User Registration and Authentication

- Users should be able to sign up and log in with email and password.

Account Management

- Users should be able to view their account details, such as balance and transaction history.
- The application should allow users to perform transactions between accounts.

User Management

- Admins should be able to view all accounts, transactions, and users in the system.

1.2.2. Non-Functional Requirements

Performance

- Optimization of resource usage.
- Provide fast response times.

Scalability

- Event-driven architecture with message brokers (such as Azure Bus or RabbitMQ) should be used for communication between services.

Reliability

- Implement good error handling to gracefully handle failures.

Interoperability

- Proper handling of asynchronous events.
- REST API for communication between backend- & frontend services.
- Make sure JSON encoding/decoding is possible.

Security

- Hashing of stored user passwords.
- Implement TLS/SSL encryption to secure the communication. This is also automatically added by the various Azure services.

Maintainability

- API and Databases should have documentation.

Usability

- Make the user experience good by focusing on a simple and logical user interface.

Portability

- Make it possible for the web application to work seamlessly on all of the up-to-date major browsers (Brave, Chrome, Edge, Firefox & Safari).

1.3. Technology Stack

The main purpose of a technology stack is to provide a clear and well-defined set of tools, technologies, and frameworks to build and run a software application. A tech stack helps to standardise the development process and ensure that all team members are using the same tools and approaches to achieve the desired outcome.

At the start of the project, we had some thoughts about whether we should choose to use 2 different languages for our backend and not just JavaScript since Java is the preferred language of one member. However, we quickly realised that we would have to prioritise the different parts of this project, due in part to changes in group members' activity and problems with our time schedules and planning. We therefore decided to stick with one language which ended up being javascript, with the node.js framework.

1.3.1. Programming Language

JavaScript:

- JavaScript is a very popular and mature programming language often used for web applications. JavaScript is a dynamically typed language and supports a wide range of programming paradigms like object-oriented, procedural, and functional programming.[1]
- The primary reason we chose to work with javascript as our main language, is very simple. It is the language that the team has the most experience with currently and therefore also the language we feel most comfortable with.

1.3.2. Frameworks

Node.js Express:

- Node.js is an open-source server-side JavaScript runtime environment.[2] Express is a fast and lightweight framework for Node.js. It provides several features for building web applications.[3]
- We chose the node.js framework specifically for the ease of creating reliable web pages and backend structure. It is also the framework we are the most comfortable with. Furthermore is it also the framework we are working with in class, and therefore the best framework for us to also keep up with our other subjects.

Svelte:

- Svelte is a JavaScript framework that has a small bundle size but is very efficient.[4]
- From the start of the project, the choice was either React or Svelte. Neither of us had much experience with heavy-duty frontend frameworks, except Rasmus, who

[1] https://en.wikipedia.org/wiki/JavaScript
[2] https://nodejs.org/en
[3] https://expressjs.com/
[4] https://svelte.dev/

had worked with Svelte. We, therefore, chose Svelte, instead of React, since our thoughts were that a little experience is better than none.

1.3.3. Data Storage

MySQL:

- MySQL is an open-source relational database management system. It is often used together with web applications because of its performance, reliability and ease of use.[5]
- We chose MySQL due to two reasons:
    - We felt that this bank system would work best with a relational database. The reason for that is that most of our models, such as customers, transactions, accounts, and more, are working with snapshots and therefore all have heavy relations with each other.
    - We have hosted and worked with MySQL databases before on AWS and Azure which made this choice an easy one.

1.3.4. Messaging & Event Streaming

Azure Service Bus:

- As we already have 2 backends and 1 frontend on Azure, plus the fact that we worked with service bus in our other subject, made us lean more to the side of the service bus, over RabbitMQ. Another factor was also the simplicity of the node.js implementation. We decided to try both RabbitMQ and Azure service bus, as a service. We decided that we liked the service bus the most, and therefore went with that.

1.3.5. Query Language

GraphQL:

- GraphQL is a query language for APIs. It is very useful for microservice architectures since it allows for gathering data from multiple sources into a single API.[6]
- GraphQL was news for all of us, which gave us the opportunity to work with something new. Until now, we primarily worked with express and REST APIs. We have used a simple version, where we translated our database tables to GraphQL schemas and used them as such.

1.3.6. Containers & Orchestration

Docker maintainability, reliability, and scalability with their way of simplifying the development and allowing us to the easier deployment of our application and upcoming updates for it.

---

[5] https://www.mysql.com/
[6] https://graphql.org/

Docker & Docker-compose:

- Docker[7] is a containerization platform that allows us to package our applications and dependencies into containers, making them easier to deploy and run consistently across different environments.
- We have decided to run our development environment (backends) in docker containers and build them with docker-compose. The production is built with GitHub actions through Azure web apps. Furthermore, we have been running our development edition of RabbitMQ, databases, and more in docker containers, until we moved past that and hosted them on Azure.

Kubernetes:

- We thought about using Kubernetes[8] with Docker containers to host our services, but in the end, we decided to go with simple web service apps instead. This was mainly due to a time constraint, where we earlier on, had problems with docker-compose that we decided we would like to avoid if possible. A lack of experience also made this task seem more daunting.

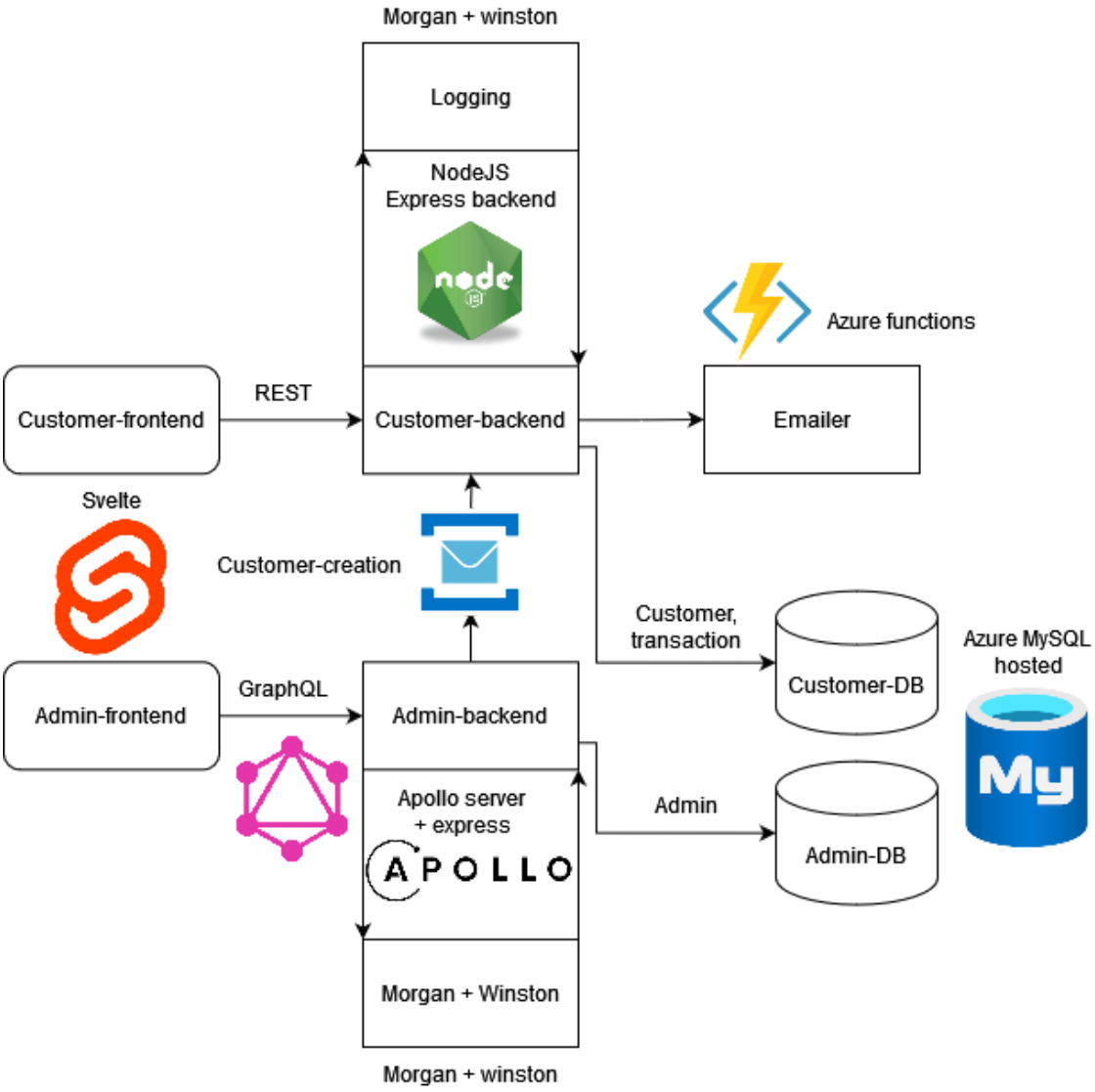
2. System architecture

We ended up using an architecture, with two frontends, two servers, two databases, and a message broker handling requests between the servers.
Each service has a docker-container, which is spun up via docker-compose in the development environment. This way, we can easily log errors, and quickly get the project up and running. For the production were using the Azure plugin for vs code, to easily deploy our different services directly to their Azure namespace. This was done with an automatic CD channel, where we simply told the plugin to deploy it, and everything was handled by Azure and a simple GitHub action, made by Azure itself. We had a “json.settings” file with configuration for the deployment, that looked like this:

It tells some metadata about the Azure namespace and its belonging data, which in this case is our customer-backend.

We wanted the customers, and the objects associated with the customers (transactions), to be saved in a separate database from the one containing data from the admins. This increases security, by minimising the risk of a fatal breach in either database, but costs more in terms of database upkeep.

[7] https://www.docker.com/
[8] https://kubernetes.io/



First microservice is the admin-backend, which has a GraphQL-API. This means the frontend queries for data using GraphQL-queries, which can send back specific attributes of objects contained in the database, without sending everything.
Admin is in charge of creating, updating and "deleting" customers (more on that later). This is because a bank system does not just allow its customers to sign up by themselves, but requires authorization. Therefore, the admin creates customer-objects through rabbitMQ, which in turn inserts these into the customer-database.
Among the admins, some are superusers. These superusers are capable of creating, updating and deleting other admins in the admin-database. This makes it possible to create an admin using the frontend, but impossible to create an admin without the right credentials.

The customer-backend uses REST-API to respond to queries from the frontend. It is in charge of creating accounts for its users, which allows the user to create a "college funds"-account or a "house accounts"-account. When the user wishes to transfer money

to another account, they create a transaction-object. When creating a transaction, two transactions are automatically created; a negative transaction, with a negative amount, which is added to the sender's history, and a positive transaction, which is sent to the account of the receiver. This is called double-entry bookkeeping, and improves detection of errors.
This system is of course too simple for the real world, and is more a proof of concept than anything. Under normal circumstances, we'd also create transactions when a customer adds money to their account.

The emailer is as simple as can be. It takes care of sending mail, i.e. when sending a transaction to a different account user. It could potentially be used for authorization, in cases where the user would like to reset their password. It is hosted via azure, and has a http-trigger.
The mailer is nodemailer, a package capable of sending emails from a premade account to another.

All objects created, both admin, customer, account, and transaction, implements the tombstone-pattern, making both databases insert-only. As an example, the admin-object admin has a created_at timestamp, deleted boolean, and a deleted_at timestamp. Then, an admin_data object, which contains all the data of the object at a point in time. This object refers to the admin-object via a foreign key, which means that we, instead of deleting an object, simply create a new admin_data object. When we need the info of an admin (or all admins), we first check whether or not the object has been "deleted", and then sort by latest created_at date of admin_data. This ensures that no objects are deleted, which eliminates the risk of deleting something crucial by accident.

We also need to ensure that the code is idempotent, i.e. performing a function multiple times will not change the result. If we do not write idempotent code, it could lead to i.e. errors, where an admin is created twice, but with different IDs. We have already partially removed all cases of dangerous put and update queries, as we use the tombstone and snapshot pattern. When "updating" an object, the customer only adds a new object to the database, meaning adding multiple only adds multiple objects with different timestamps, risking nothing.

2.1. Microservice 1 (Mysql databases)

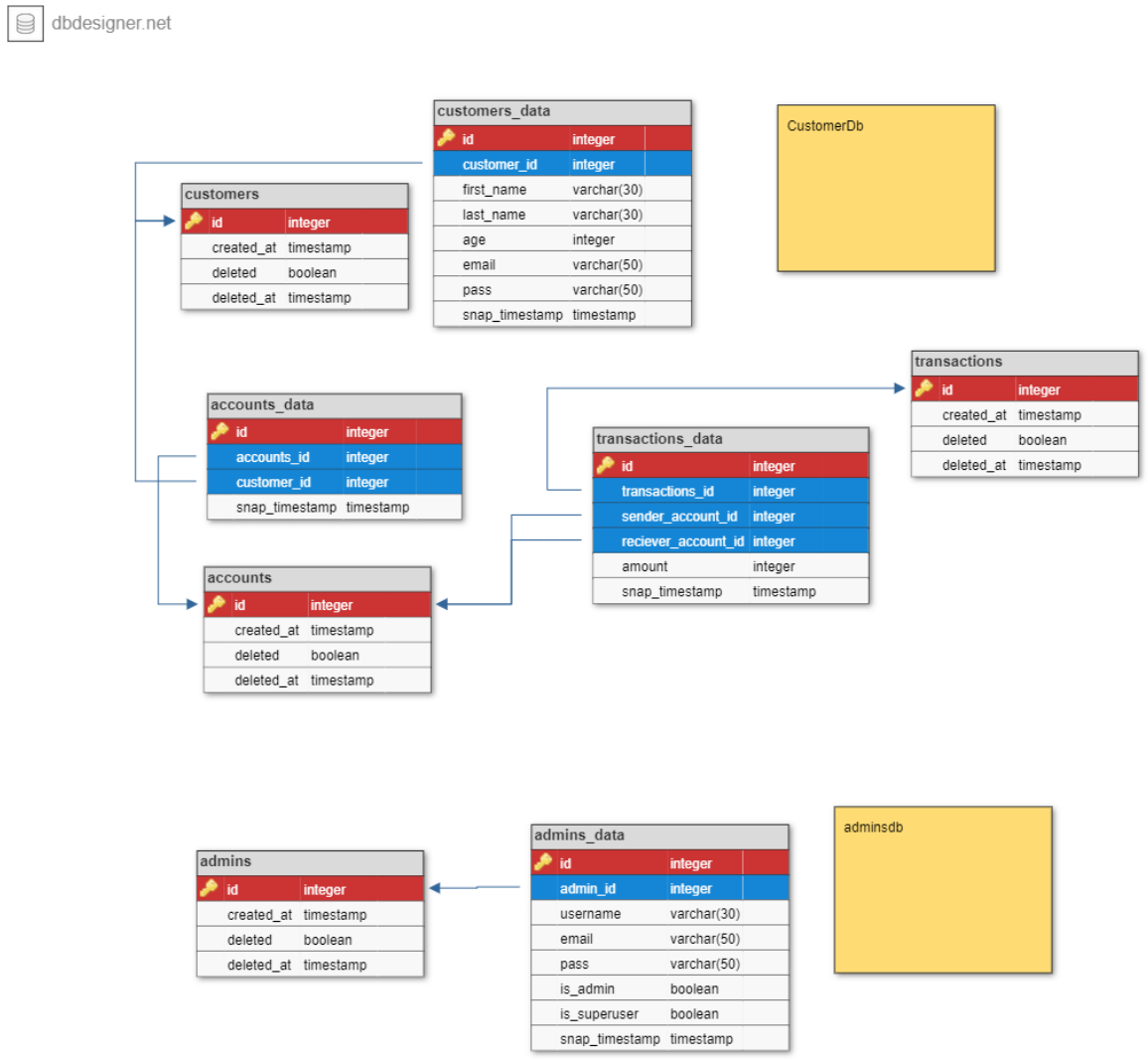
We are running two MySQL database servers on Azure. One for customers and their related entities and one for admin. We thought a lot about the pros and cons of splitting up the databases instead of simply having one. We knew that we would not be able to create a big and complex system, with all of the problems we have had in the development, and furthermore would the entities in the database not be filled with very

much. So why would we spend the resources and manpower on setting up and using 2 databases instead of simply using one database for all of it.
We decided that we would stick as close as possible to the functions of a real bank and at least use the most basic features. One of these features, we felt, was a sensible separation between a bank's users, both in terms of simplicity but also future security conditions. It was ultimately this that made us choose 2 databases as the way we wanted to go instead of one database for all our data.

We chose to make use of Azure's free services as much as possible, and therefore also chose to use their "Azure Database for MySQL flexible server". We have worked with AWS before but decided to go with Azure, because of two things; we have a student account, which makes it free for us to use (as long as we do it correctly) and we have been working with Azure in the SI course.
For ease of use, we decided to leave the database open to the outside world. We felt comfortable doing this since we don't really have any sensitive data in these services and what we were primarily nervous about was the use of our student credit, which we kept an eye on, via Azure's built-in "spending-alert", where we could stop services if the consumption became excessive. For these reasons, we were comfortable providing the database "public access".

Our customer database consists of *customers, accounts,* & *transactions*. Each of these is then separated into its base version and its _data version, which is our way of handling snapshot patterns. We will go more in-depth with this later on.

Below is the design of our databases.



As visualised in the model, are our databases fairly small and with a heavy amount of relations between each other, which is also one of the reasons that we chose a relational database like MySQL.

2.2. Microservice 2 description (Customer & Admin backends)

Our project has 2 backend services, admin-backend, and customer-backend. Their name tells a lot about their responsibilities. The customer backend is responsible for the creation and manipulation of customer-related entities, such as accounts and transactions. It is also this service that is connected to the customer database.

Customer backend

The customer backend is using the express framework with a REST API. We decided that we wanted the customer-backend to be responsible for the communication with the customer database, and all the manipulations therein. The way we did this was by

separating each entity in the database with its own router. Each of these endpoints is well documented with swagger, which we will be talking about later on.
We have decided that our code flow should be handled with a promise based async/await flow. We did this, due to multiple factors, the main ones being that we simply like promise based flows better, it's more readable and we had times where we had some flow interruptions with just using a callback flow, that got fixed after we switched it to await. That is of course not to say that we have not used callback functions, as we use it in all our express endpoints. As seen in this figure.

```
router.get("/customers", authenticateToken, async (req, res) => {
    try{
        const result = await getCustomers();
        logger.verbose("the result recieved from getCustomers function", result);
        res.status(200).json(result);
    } catch (err) {
        logger.error(err);
        res.status(500).json("Internal server error");
    }
});
```

The above mentioned image visualises the approach to end point creation we have used, both in customer and admin backend. We have the endpoint, made as a callback function, that has an asynchronous flow inside.
The way we have made our endpoints, is that we separate the internal logic into other functions, that is then just called from the endpoint.
There are multiple reasons for us doing this. Our logic is to separate the execution from the point of access, and allow other parts of the code to access the code without having to make a http request, to the given endpoint. Below is the related function for the endpoint shown before.

```
export async function getCustomers() {
    const connection = await conn.getConnection();
    try {
        let [rows] = await connection.query('SELECT * FROM customers c JOIN customers_da
        connection.release();
        rows = JSON.parse(JSON.stringify(rows));
        return rows;
    } catch (err) {
        logger.error(err);
        connection.release();
        throw err;
    }
}
```

The authenticate function in the endpoint is part of our JsonWebToken security, which we will talk about later in the report.

Admin backend

The admin is responsible for the creation of customer entities from the front end. Our thought was that we wanted to split up the responsibility of customer creation from the customer since you usually are not able to manipulate or create either yourself or other users in a bank system. Therefore, the logic is located in the admin backend. It uses GraphQL for the communications between the admin- and customer backend. It also uses the Azure service bus, for all actions involving creation and manipulation of customers. For the GraphQL part, are we using an Apollo server to handle the general routing in the admin backend.

The way we handled this was by creating a schema for the different actions we wished to be able to do. Our approach for this was to follow the general CRUD pattern. Where we primarily used *query* for the GET endpoints and *mutation* for the POST parts. This is an example of our schemas.

```
type Mutation{
    Login(username: String!, pass: String!): AdminLogin
    CreateCustomer(firstname: String!, lastname: String!, age: Int!, ema
    UpdateCustomer(customer_id: Int!, firstname: String!, lastname: Stri
    DeleteCustomer(customer_id: Int!): CustomerCombined
}
```

```
type Query {
    # Admins
    GetAdmins: [AdminCombined]
    GetAdminById(admin_id: Int!): AdminCombinedById
    GetDeletedAdmins: [AdminCombined]
```

It is worth mentioning that even as we have used both *query* and *mutation*, did it all require that we used a post method for all of our calls to the GraphQL API, as it requires the query to be sent with the requests, as that is the way that GraphQL defines which API endpoints/resolver it has to route to.

We build our resolvers to take data from the front end, and then send it to relevant functions that then is responsible for handling the execution. An example is the login/authentication of admin users.

```
Mutation: {
    Login: async (_, { username, pass }) => {
        try {
            const msg = {"username":username, "password":pass}
            const loginData = await loginAdmin(msg);
            console.log(loginData.msg)
            return loginData.msg;
        } catch (err) {
            logger.error(err);
        }
    },
```

It receives the username and password, which are then sent to the authentication function that returns the user, with a JsonWebToken, which is then returned to the front end and used to access other pages.

Due to the fact that most of the actions in our system is done in the customer backend, is this the smallest of the two services, but might just be the most complex one, with GraphQL implemented and the responsibility for the customers

2.3. Microservice 3 description (Service bus)

We have decided to use an Azure service bus as our message broker. We will talk more about the service bus and our choice of communication later in this report.

2.4. Microservice 4 description (Email-service / serverless functions)

As part of the creation of a new customer in our bank, will each customer receive a welcome mail with the possibility for added features in the future. Since this is something that can be done, inside the scope of a simple function, was it our decision that this should be placed inside of a serverless function. The reason for that was twofold. On one hand, would it enable us to easily use the function for both admins and customers if we decided that it would be relevant for the admin too. Since we decided that it should work as a courtesy mail and not have user-specific information (besides the name of the customer), was it our decision that it did not need to be used for admin too.
Furthermore, could this service also be used to make it possible for the customer to get in contact with the bank, after some reviews on this feature, was it our consensus, that it should be something that would fit poorly when looking into how banks normally do contact from the customer side.

To do this, we used Azure's *Functions* service, which we deployed as a simple code version. That is, done through the Azure plugin in visual studio code. In this way, are we letting Azure handle det deployment of the service, we simply make changes to it in vs code, and deploy it to Azure.

We have set up a new Outlook mail, that is used as the bank mail, and put as the sender of the mail. We chose to use Nodemailer as our email client, and Outlook as our main SMTP service to target Outlook. The reason we chose to target Outlook is fairly simple. Gmail and others have higher scrutiny and do often not allow emails from services like Nodemailer and such. That is not the case with Outlook and why we chose to use that.

2.5. Microservice 5 description (Svelte)

Our front end employs Svelte, a framework with a focus on speed and efficiency, due to it compiling what is presented to the browser, instead of putting the bulk of the work on the browser. As Svelte is great for building single-page applications (SPA), it allows us to create efficient routing, and add secure conditionals; the conditionals in the HTML are compiled in the backend, and dynamically updated in the DOM. This means the user cannot access data, until the conditionals are met, and the HTML is switched. It also allows us to create components, which can be easily applied to the different pages, such as a topbar.

2.6. Communication between microservices

The main channels of communication between our services are Axios/fetch (HTTP requests) and Azure service bus (message broker). The service bus is used for communication between backend services, specifically interactions with a customer, which we have decided should lay solely in the hand of admins, as you usually can not change your own information in a real bank either.
Axios is used for communication between frontend and backend, and occasionally for smaller services like our email service.

2.6.1. Axios & Fetch

2.6.1.1 Frontend to backend communication

We made the decision that the main way of communicating with the backend from the frontend would be through API-calls with Axios. It would probably be possible to do this part with message queues and azure service bus too, but we made the decision that we would use Axios and fetch instead, due to the fact that we ended up with time constraints that felt unreachable if we did not use these.
We are using different types of Axios and fetch requests. Those to express endpoints and those to graphql endpoints. Below is an example of one of these endpoints. In this case is

a fetch to the GraphQL API, that creates a customer with the help of the Azure service bus in the admin backend.

```
const response = await fetch('https://dls-admin-backend.azurewebsites.net/graphql', {
    method: 'POST',
    headers: header,
    body: JSON.stringify({
        query: query,
        variables: variables
    })
})
```

In this example, we are fetching the GraphQL endpoint, which is shared by all our different schemas in the resolver as mentioned earlier in the explanation of the admin backend. We will therefore not go into many details in this section.
We found that fetch was much easier to use when working with GraphQL, and Axios was preferable when working with express or that is at least our opinion. Here is an example of that usage.

```
await axios({
    method: 'post',
    url: 'https://customerbackend.azurewebsites.net/auth/login',
    headers: {
        'Access-Control-Allow-Origin': '*',
        Accept: "application/json",
        "Content-Type": "application/json;charset=UTF-8",
    },
    data: data
})
.then((data) => {
    if (data.error) {
        alert(data.error);
```

2.6.2. Azure service bus

As mentioned elsewhere in this report, we spent a lot of time figuring out which of the 2 types of message queue services we would use. We decided that we would either use RabbitMQ, where we set up a RabbitMQ server, through their own service, or Azure service bus.

Why the Azure service bus and not RabbitMQ?
We definitely had the best experience with the Azure service bus, and therefore also the one we ended up choosing. One of the biggest reasons why we chose the Azure service bus over RabbitMQ was the increased availability and simplicity of the Azure service bus. We decided that we wanted to use promise-based communication rather than, for example, callbacks, which we also felt Azure supported best.

Usage of service bus
As mentioned in our explanation of our backends, did we decide to use the service bus when communicating between the two backends in relation to customers. We had plans to implement and use a service bus between the back and front end too, which sadly had to be deprioritized.

We chose to make use of an RTS progress, where we created 2 queues, one for requests and one for responses. So when you from the admin backend wanted to do something with the customer backend, you would send a message up in the request queue, which the customer backend actively listens to. The customer backend will then handle the request from the admin backend, and respond through the response queue.

Name	Status
customercm	Active
responsecm	Active

The way we handle messages sadly does not reach the level of *Commutative message handlers*, since the way we have set up the messages in the RTS process, uses the FIFO principle, where the first message sent to the queue is also the one that gets consumed first. The cons of this are clear. If an error happens at the customer backend when consuming a request but before it makes a response, then we can risk that the system manages to create or manipulate a customer, but not make the admin backend aware of this, which might throw an error in the admin. To minimise the risk of such a situation, have we created a robust path for a message to take, with clear requirements. We need to get precise information from the admin before we can make a request message. That information is exactly what is needed for customers. Then when the response comes back will it always be in the form of a JSON object or array. One of the main selling points of GraphQL which is what we use in the admin backend is the diversity of what you can request. Here we ensured that the data needed in the customer backend is required to be sent.

One of the weaknesses our message queues have when we use RTS as we do is when unseen accidents stop or close our program. There, our messages will be caught in the queues, and therefore also block other messages in our queues, due to our use of FIFO.

To get rid of this error, we set a message expiration time of 15 seconds. That's more than enough to be sent back and forth between our backends.

Since we use GraphQL, we can be sure of what is being sent and can therefore use resolvers to make checks in our service bus, so that we can specify which actions we want to do. Below is an example of how we have chosen to handle this.

This is an example of the resolver in the admin backend, that gets a specific customer from the customer backend. As part of the message, we set a variable called typeOfMessage which helps to tell the service bus in the customer backend which action we want to perform.
In this case, it is "readSingle" that tells the customer backend that we want a specific customer based on the customer_id we send in the body.

```
GetCustomerById: async (_, {customer_id}) => {
    try{
        const msg = {typeOfMessage:"readSingle", body: {id: customer_id}};
        const result = await sendToQueueFunc(msg);
        console.log("THIS IS THE RESULT: ", result);
        return result;
    }catch(err){
        console.log(err);
    }
},
```

Then we send that JSON to the function called *sendToQueueFunc*, which is the service bus function that is responsible for the communication with the customer backend.
That function defines the URL of the Azure namespace, the name of the different message queues, and then sets the values from the GraphQL resolver into the requests of the message, as seen in this figure.

```
const requestMessage = {
    body: bodyValue || "error sent",
    applicationProperties: {
        bodyType: msg,
        priority: 1,
    },
    replyTo: responseQueueName,
    messageId: Date.now().toString(),
    contentType: "application/json",
};
```

The requestMessage constant shown above is the message we are going to send to the service bus request queue. The body is the body part of the msg from the resolver. It is the data that is going to be used to make database queries in the customer backend. We then use the applicationProperties to set a custom value called bodyType. This is the specification for what type of action we wish to use in the customer backend. It is the *typeOfMessage* from the graphql resolver.
Next up is the replyTo, which is part of the RTS process we are using. This part tells the customer backend which message queue we will be looking for the response. Then we set the messageId to be a Date.now(), since we know that it will change every second to a different integer value, and therefore effective to use in this case, as we always want a different value for each message, as this is the correlationId.

Then on the customer backend side of the service bus, we got this:

```
}else if(requestMessage.applicationProperties.bodyType === "readSingle") {
    console.log("TYPEOF : read-single")
    responseMessageBody = await getSingleCustomer(requestMessage.body.id) || "Error with
```

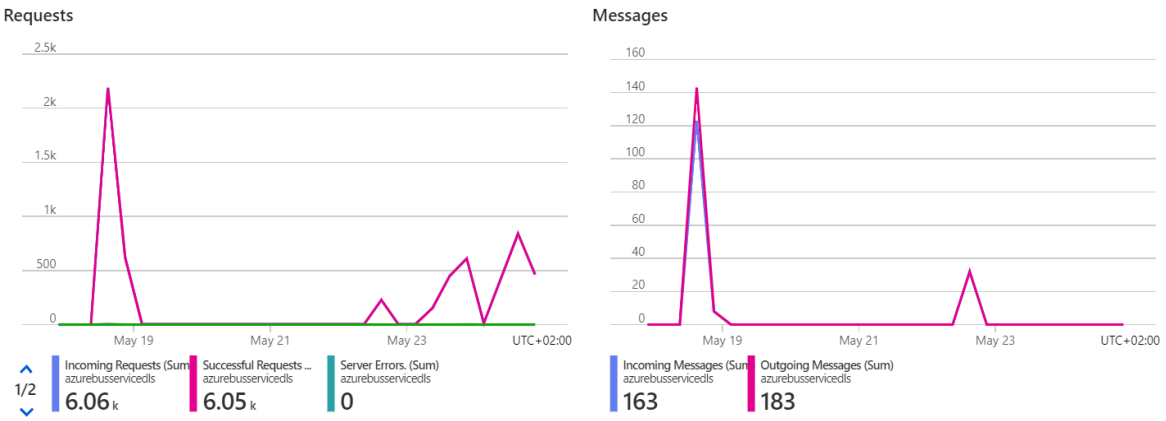
This is where we use the *bodyType* value we saw earlier in the requestMessage. We then simply call the related function, which in this case is the *getSingleCustomer*.

```
const responseMessage = {
    body: responseMessageBody,
    correlationId: requestMessage.correlationId,
};

// Send the response message to the replyTo address
const responseSender = serviceBusClient.createSender(requestMessage.replyTo);
await responseSender.sendMessages(responseMessage);
```

The responseMessageBody receives the result in json format, and uses it to create the const responseMessage together with the correlationId (messageId) from the request that was consumed in the beginning. We then send the message to the message queue that the admin backend told us it wanted the customer backend to send the reply to.

One of the great things about hosting services on azure is its easily manageable monitoring and logging, built into the different services. One of these capabilities that we used a lot, was the simple but effective overview of usage and active message as seen below.



The same is applicable to each queue, where we quickly can get an overview of the amount of active queues, which queues are waiting to be consumed and much more.
The simple and easily understandable overview, was something that really separated the azure service bus from the seemingly old looking UI in rabbitmq. We were on the other hand, able to get rabbitmq to automatically create and then destroy queues directly from the code, which was a feature we missed from the azure service bus.
Furthermore, was it much easier to implement azure service bus in a promise based code, where rabbitmq seemed to prefer to do callbacks. Since our backends are built upon a promise based async/await instead of callbacks, was this a clear win for azure service bus.

If our time schedule was not moved as much as it used to be, would it have been a great benefit for us to also use azure service bus in the communication between back- and frontends, which we felt we had to deprioritize this time around.

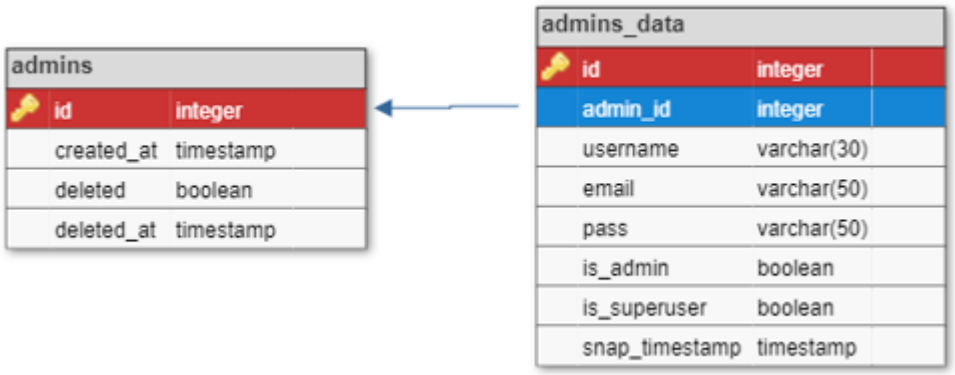
2.7. Description of the patterns and techniques used in the project

2.7.1. CQRS

In our project, we do not implement the classic architecture of CQRS. We have a database to take care of both read, write, update and delete. To create less traffic, improve scalability, and avoid fatal database-manipulation, it'd be more secure to create a second read-only mirrored database of our main database. This comes with the cost of extra upkeep, and extra complexity, in exchange for the aforementioned security. As such, it seems it'd only be necessary in the backend of our customer service, as it has the most traffic.

2.7.2. Immutable data (Tombstone pattern and Snapshot pattern)

We have implemented tombstone and snapshot patterns in our database for all of our entities. As mentioned earlier, are each of our entities separated into two parts, the "base" part and the "data" part. This separation is show in this figure:



2.7.2.1. Tombstone

We have an admins table that consists of a timestamp that defines the time of creation and the part that defines the tombstone pattern. It is a simple boolean that describes whether or not the admin is active or inactive and when the change occurred.

We use the tombstone part of the admins table, in all of our database queries where we check if the "deleted" is false. In that way we ensure to separate off the admins that are no longer active. An example will be shown and explained in the snapshot part.

2.7.2.2. Snapshot

The separation of admins and admins_data table, shown in the earlier picture, is the way we decided to handle the snapshot pattern. The way this works is that there is a one to many relationship between admins and admins_data. Instead of deleting data about an admin, we instead make a new admins_data with the id of the connected admins entity as admin_id. The way we ensure to always use the correct version of admins_data is by sorting by the "snap_timestamp" field. This way we do not need to be worried about the amount of admins_data, as we always get the newest one because of how we handle the queries to the database. This example shows how we use tombstones and snapshots patterns.

```
SELECT * FROM admins a JOIN admins_data ad ON a.id = ad.admin_id WHERE
(ad.admin_id, ad.snap_timestamp) IN (SELECT admin_id, MAX(snap_timestamp) FROM
admins_data GROUP BY admin_id) AND a.deleted=false;
```

This query takes all active admins with their latest admin_data. The yellow part is the section where we use the tombstone part. Here we tell MySQL that we only want the admins, that are not deleted.

The green part is where we use the snapshot pattern part. We tell MySQL to look into the admins_data id and timestamp and select the admins_data with the highest value on the snap_timestamp column. As this field is simply a very high int value, ensures that we will always get the newest version since the later the date is, the higher the value.

2.7.3. Idempotence

The simplest way to avoid accidental errors caused by too many object creation requests, is to make the HTML elements inactive for a short while.
We have not implemented a perfect idempotence, as this was not prioritised.

3. Deployment

3.1. Introduction to the cloud deployment

Cloud deployment has become increasingly popular due to several benefits, such as scalability, flexibility and cost efficiency.
As a way to increase potential scalability we wish to host our services. This, of course, allows us to scale easily, as we simply pay more to get more server space, ensure accessibility, and get us away from locally hosted services and the potential hazel of big docker-compose files, which is something we had problems with earlier on in the development.
Throughout this chapter and the chapters about microservices, we will describe which technologies have been used for cloud deployment, our CI/CD pipeline and monitoring & logging of the system.

3.2. Description of used technologies

We use an Azure bus to send messages between services. The fact that the service is in the cloud makes it scalable and has a built-in logging system, which makes it useful in keeping an overview of the messages sent between servers.

Most of our services are run on Azure. Our servers run on the Azure App Service, which has automatic scalability if you pay more.
The simplest technology is Azure functions, which we use to listen for an HTTP-trigger, and then send a mail to the created user.
Our databases are running on Azure MySQL Database. The automatic server maintenance and security make it a sensible choice, along with the ability to automatically expand.

3.3. CI/CD pipeline description

Our CI/CD pipeline is very rudimentary.
We use GitHub to pool our project and its branches. Github has a built-in option for a pipeline, called GitHub Actions. This trunk-based development fits our group well, as we are a small group, and none of us are unaware of what the others are doing but minimises accidental implementation of errors. We'd also like to always have the newest version of the project, so as to not implement something, which has already been covered.
In a business setting, we'd like to have unit tests in our pipeline, checking our code and making sure it is consistent. Preferably an automatic system, where the unit tests are written directly in the code, and picked up when pushed to the main branch.

3.4. Monitoring and logging of the deployed system

We decided to use Morgan and Winston for logging because of our previous experience with them. Morgan and Winston are both logging libraries that are very well-documented and easy to implement and use.

Apache has defined several log format standards and we have decided to use the (Combined Log Format)[9], which is a little more detailed than the "common" log format. The combined log format also logs information about the user's type of operating system, processor architecture, and type of browser. This will be demonstrated later in this chapter.

```
1  import morgan from "morgan";
2  import logger from "../utils/logger.js";
3
4  const stream = {
5    // Use the http severity
6    write: (message) => logger.http(message),
7  };
8
9  const skip = () => {
10   const env = process.env.NODE_ENV || "development";
11   return env !== "development";
12 };
13
14 const morganMiddleware = morgan(
15   // Define message format string (this is the default one).
16   // The message format is made from tokens, and each token is
17   // defined inside the Morgan library.
18   // You can create your custom token to show what do you want from a request.
19   //":remote-addr :method :url :status :res[content-length] - :response-time ms",
20   //":remote-addr - :remote-user [:date[clf]] :method :url HTTP/:http-version :status :res[content-length]",
21   //":remote-addr - :remote-user [:date[clf]] :method :url HTTP/:http-version :status :res[content-length]",
22   ":remote-addr - :remote-user [:date[clf]] :method :url HTTP/:http-version :status :res[content-length] :referrer :user-agent",
23   // Options: in this case, I overwrote the stream and the skip logic.
24   // See the methods above.
25   { stream, skip }
26 );
27
28 export default morganMiddleware;
```

As seen in the picture above, Morgan is used to define the HTTP messages of Apache Combined Log Format. The Combined Log Format string has been added to line 22. If let's say we want to change the log format it can simply be done in this method. For

[9] https://httpd.apache.org/docs/2.4/logs.html

example on line 21 we see the commented Apache Common Log Format, so this can easily be done by un-comment that line and commenting out the current line 22.

According to the Syslog Protocol(Internet standards track protocol), **RFC5424**[10], log warnings should be categorised based on severity. The lowest number, 0 defines the most severe message, and the highest number defines the least severe message.

```
1   import winston from "winston";
2
3   const levels = {
4       error: 0,
5       warn: 1,
6       info: 2,
7       http: 3,
8       verbose: 4,
9       debug: 5
10  };
11
12  const level = () => {
13      const env = process.env.NODE_ENV || 'development'
14      const isDevelopment = env === 'development'
15      return isDevelopment ? 'debug' : 'warn'
16  }
```

We have defined the logging severities by using the same standard as mentioned in the Syslog protocol. The error code has a level of 0 meaning it is the most severe type of message, while level 5 “debug” is the least severe type of message in our system.

[10] https://datatracker.ietf.org/doc/html/rfc5424#page-11

```
18  const colors = {
19      error: 'red',
20      warn: 'yellow',
21      info: 'green',
22      http: 'magenta',
23      verbose: 'cyan',
24      debug: 'white'
25  };
26  winston.addColors(colors)
27
28  const format = winston.format.combine(
29      winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss:ms' }),
30      winston.format.colorize({ all: true }),
31      winston.format.printf(
32          (info) => `${info.timestamp} ${info.level}: ${info.message}`,
33      ),
34  )
```

To make monitoring easier for us we have colorised each log message level. As an example, errors will shown in red text in the terminal, while http info, like get requests, will be shown as magenta, which will be demonstrated later in this chapter.
Line 26 tells Winston to use the colours defined in the colors object above. From line 28 till 34 we use Winston to format the logs with timestamps, colors, info and messages.

```
67
68  const logger = winston.createLogger({
69      level: level(),
70      levels,
71      format,
72      transports,
73  })
74
75  export default logger
```

The code above wraps all the constants we have defined earlier into a logging constant, which we can call, whenever we wish to log something in our program.
For example, if we want to log an error, we can simply call logger.error(err) with the “err” parameter or logger.info() if we want to log system-wide information.

In the picture above we can see the different colours that are being printed in the terminal for easier distinguishing between the different severities, which makes monitoring logs in the terminal easier. This picture shows HTTP requests made to the server, which in this case are GET requests including data like the IP address, the date and the time of the request, the user's type of operating system, processor architecture, and the type of browser used for the requests.

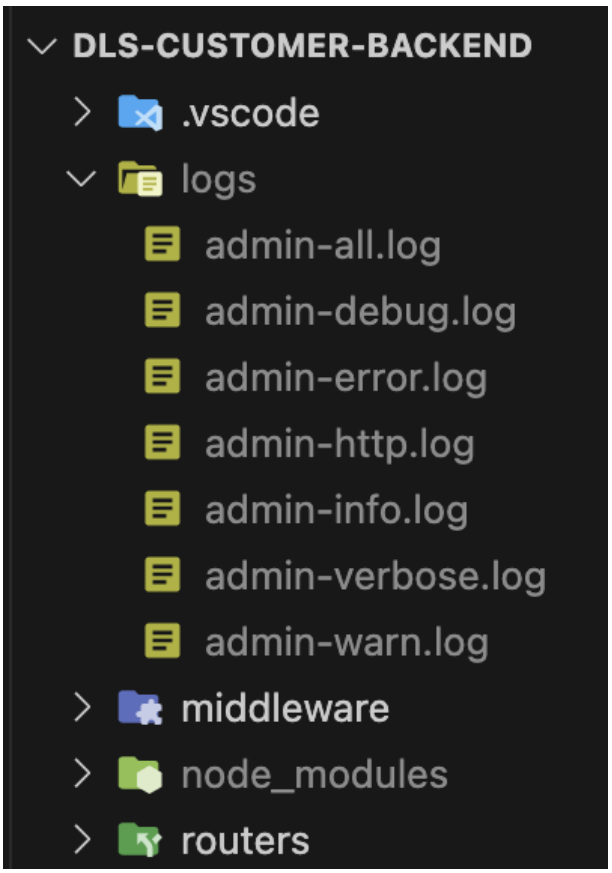
```
21  async function testDBConnection() {
22      const connection = await pool.getConnection();
23      try {
24          logger.info("Connected to database");
25      }
26      catch (err) {
27          logger.info("Could not connect to database");
28          process.exit(1);
29      }
30  }
```

Above we can see a try-catch to check if the application can connect to the database. Here we log simple info on whether it could connect or not.

```
36  const transports = [
37      // Allow the console to print messages
38      new winston.transports.Console(),
39      // Print all the error messages inside the error.log file
40      new winston.transports.File({
41          filename: 'logs/admin-error.log',
42          level: 'error',
43      }),
44      new winston.transports.File({
45          filename: 'logs/admin-warn.log',
46          level: 'warn',
47      }),
48      new winston.transports.File({
49          filename: 'logs/admin-info.log',
50          level: 'info',
51      }),
52      new winston.transports.File({
53          filename: 'logs/admin-http.log',
54          level: 'http',
55      }),
56      new winston.transports.File({
57          filename: 'logs/admin-verbose.log',
58          level: 'verbose',
59      }),
60      new winston.transports.File({
61          filename: 'logs/admin-debug.log',
62          level: 'debug',
63      }),
64      // Print all the error message inside the all.log file
65      new winston.transports.File({ filename: 'logs/admin-all.log' }),
66  ]
```

In the transport array above, we first allow Winston to print log messages to the terminal, which is done in line 38.

From line 40 to line 66 we use Winston to automatically generate log files for us. If the log files have been backed up and then cleaned, to make the overview easier, Winston will then keep creating these log files for us in the /logs/ folder. The log files, from lines 40 to 63, will log messages related to their given level, whereas the all.log file created in line 65 will simply log every message in the system.



In the earlier example, we showed how we autocreate these logfiles with the Winston middleware. In the picture we can see the folder “logs” which contains all the log files that we defined earlier. These are being stored on the application’s file system.

```
// GET ALL ACCOUNTS
router.get("/accounts", async (req, res) => {
    try {
        const result = await getAccounts();
        logger.verbose("the result recieved from getAccounts function", result);
        res.status(200).json(result);
    } catch (err) {
        logger.error(err);
        res.status(500).json("Internal server error, or no accounts found");
    }
});

async function getAccounts() {
    try {
        const connection = await conn.getConnection();
        let [rows] = await connection.query('SELECT * FROM accounts a JOIN accounts_data ad ON a.id = ad.account_id;');
        connection.release();
        return rows;
    }
    catch (err) {
        logger.error(err);
        connection.release();
        throw err;
    }
}
```

In the code above for getting all accounts, we use logger.verbose() to first log the info message, then pass on the “result” from getAccounts() method.

On lines 15 and 28, we log the error again from the catch.

```
19          const select_customer = 'SELECT * FROM customers c JOIN customers_data cd ON c.id = cd.customer
20          connection.query(select_customer, [req.body.email], function (err, result) {
21              if (err) {
22                  logger.error("Error executing the query: ", err);
23                  connection.release();
24                  throw err;
25              }
26              if (result.length === 0) {
27                  res.status(404).send("customer not found");
28                  connection.release();
29              } else {
30                  logger.verbose('customer with name: ' + req.body.first_name + ' selected!\n ', result)
31                  bcrypt.compare(req.body.password, result[0].pass, function (err, result1) {
32                      if (err) {
33                          logger.error(err);
34                          connection.release();
35                          throw err;
36                      }
```

The picture above shows the connection query for our customer table. If an error occurs the logger.error(err) will write the error code to the error.log file before the connection is released.

```
22  async function createTransaction(values) {
23      const connection = await conn.getConnection();
24      try {
25          await connection.beginTransaction();
26          logger.info("entered transaction");
27          const create_sender = 'INSERT INTO transactions_data (transaction_id, sender_accour
28          const create_reciever = 'INSERT INTO transactions_data (transaction_id, sender_acc
29          // ----- Create transaction for sender -----
30          const sender = await connection.query('INSERT INTO transactions () VALUES ()');
31          const senderData = await connection.query(create_sender, [sender[0].insertId, value
32          logger.verbose(senderData);
33          // ----- Create transaction for reciever -----
34          const reciever = await connection.query('INSERT INTO transactions () VALUES ()');
35          const recieverData = await connection.query(create_reciever, [reciever[0].insertId,
36          logger.verbose(recieverData);
37          // ----- Commit changes -----
38          await connection.commit();
39          return "Transaction created";
40      } catch (err) {
41          logger.info("rolling back transactions");
42          await connection.rollback();
43          throw err;
44      } finally {
45          connection.release();
46      }
47  }
```

The picture above shows the transaction function for transferring money between two accounts. As seen in line number 26, 32, 36, and 41 we have different logging levels, “info” and “verbose”.

Info in this case is passing simple information, like whether the transaction has been created or rolled back. Here, the verbose level is logging both the (senderData) and (receiverData).

```
10  // Login router
11  router.post("/auth/login", async (req, res) => {
12      logger.info("Login request received");
13      logger.verbose(req.body);
14      conn.getConnection(function (err, connection) {
15          if (err) {
16              logger.error("Error connecting to the database: ", err);
17              throw err;
18          }
```

In the picture above on lines 12 and 13, we can see the logger first logs a string saying the request has been received after the login post routing has been invoked. Afterwards, the logger with the verbose severity logs the body of the request.
On lines 15 to 18, we implemented an if statement with the error parameter from the getConnection() method. Inside this if statement on line 16, the logger with the "error" severity logs the connection error to the error.log file.

4. Project management and team collaboration

4.1. Introduction to the project management and team collaboration

Having focus on great project management and team collaboration are essential for having a successful project. We have used different tools and techniques which we thought would increase our collaboration and provide a better workflow.
In the next subchapters, we will elaborate on the different methods, versioning strategies and documentation strategies we used during the project.
We've had multiple members unable to contribute, and as such, the project has mainly been built by two.

4.2. Description of the methods used during the project

In the group, we've been meeting physically whenever possible, to minimise time wasted interrupting each other in a voice chat. However, due to a rather frayed schedule, we've had to meet whenever it was possible for the members, physically or not. One of the ways we kept up with each other's progress was to give each other a short recap of our current tasks and issues, as a lighter version of a stand-up meeting.
For a quick overview of the project and its progress, we used Clickup[11], which is a virtual scrum board. Here each of us could create new tasks for the group, assign them, mark them as to-do, in progress, pending (if creation depends on another part of the project), and finished. This was a great way to remove wasted time communicating with other members about which tasks needed to be done, and who was responsible for which part of the project.

[11] https://clickup.com/

4.3. Versioning strategies for the source code, databases, and APIs

For source code versioning we decided to use Git since it is very robust, mature, and well documented, but also individually our first choice tool for source code versioning. With Git we can track changes, create branches for new features or bug fixes, and merge code changes. [12]
In addition to Git the source code is stored on Github, which allows us to easily collaborate and do code reviews, and having a continuous integration pipeline.[13]

4.4. Documentation strategy

We decided to use Swagger[14] as the tool for documenting our API endpoints. Swagger helps to provide a clear understanding of authentication, request & response structures but also which API endpoints are available and which data types are accepted at a given endpoint.
Swagger has also helped those group members who joined the project later to able to understand the endpoints easier, than if they had to manually go through the code base as the only option.

Just like code reviews, we have done reviewing throughout the project of all of our documentation to keep it up to date with our codebase. This is done because not only does the code change throughout the project, but also the requirements since maybe some implementations have been left out, due to time constraints or high difficulty.

5. Conclusion

5.1. Advantages and challenges

Our project is quite small, which makes it easy to handle and change in production. However, due to having our two backends and their databases separated, and to only communicate via Azure bus, we've had to create (and protect) quite a few endpoints.

Due to the constraint of only being two members, the project has been quite a challenge. Whenever we've split the tasks, the other members have either been unresponsive, or too sick to be able to participate, meaning the tasks eventually needed to be solved by us anyway which resulted in us not having a concrete plan for the development, as we had to split the manpower of 2, for a project planned for 4-5 people. It would not have been as big of a problem if it had been in the early process of development. The biggest problem was that the missing group members did not just tell us they could not or would not do the work, but actively said that they would do things, and then not do it.

[12] https://git-scm.com/
[13] https://github.com/
[14] https://swagger.io/

When it was then expected to be completed, which meant that the rest of the group had to do it, with a slimmer time schedule.

5.2. Pros and cons of used patterns

The tombstone pattern made it easy to avoid the most common idempotence issues, as many queries no longer updated or deleted objects, but also made the remaining queries harmless if repeated, such as an update of a customer's info, with only the timestamp changed.
We did not use CQRS, which of course would put a strain on our database in a larger project. We would've liked to have a database copy purely with readable data, to ensure minimum traffic.

5.3. Scalability

All of our services are hosted, meaning the space needed to adequately run the server is adjusted automatically, and server maintenance is not a concern. With the various patterns (tombstone, snapshot), we ensure the ability to scale up the service database-wise, and with the low dependency, it's easier to scale the project by implementing new microservices.

5.4. Possible improvements

First and foremost, perfect idempotence would be ideal. With our database structuring, we avoid many cases where not implementing the pattern could be an issue. We do, however, not take all the insert-into queries into account, and as such are not immune to accidental duplicates.

If given more time, we'd like to implement CQRS. It's simply a good idea, as it would create a more secure and scalable solution for our project. At its current size, our project would not be hurt much by the increased complexity, and would be one of the last big implementations of patterns for our database.

A logging program, running on an external server, would have been nice. Connected to our various resources, it would give a great overview of object creation, and server messages; both for diagnostics and as a map of the outputs of the program through time. The current logging files function well enough for development, but better logging would have been great for diagnostics during deployment.

In the long run, the SAGA pattern would have made sense. With the amount of microservices we deploy, and the amount of messages and queries sent back and forth, a failsafe in cases of multiple transactions failing at one point or another would have been good for architectural sturdiness.

For large distributed systems caching for example, Redis could be really beneficial for user performance, since it is an in-memory database that can be used to cache the data, which is most often being fetched by the user.

6. Repositories and user-guide

6.1 Repository & deployed services

Github repositories:
https://github.com/RaskoeStud/dls-admin-backend
https://github.com/Omoezone/dls-customer-backend
https://github.com/RaskoeStud/admin_frontend_svelte
https://github.com/RaskoeStud/customer_frontend_svelte

Deployed Applications:

(frontends)

(admin) https://ambitious-ground-0000e7c03.3.azurestaticapps.net/
(customer) https://agreeable-desert-081cff403.3.azurestaticapps.net/

(backends)

(admin) https://dls-admin-backend.azurewebsites.net/
(customer) https://customerbackend.azurewebsites.net/

6.2 Setup and documentation

It is possible to access swagger documentation for each backend service, using the /api-docs endpoint. We currently do not have any separated dev environment, besides running the different services on localhost. The databases are all in the cloud and are no longer accessible locally. To be able to use the backends, is it required to use these two .env files. They should each be placed in the project's root folder.

Refer to the environment variables in the “environment variables.pdf“ appendix.

6.3 How to build and run services

The two backend services do not require any building and can be run in a local environment with the node command “npm run start-dev”.
The admin backend is set to run on localhost:3000.
The customer backend is set to run on localhost:5050

The Svelte frontends do require to be built first, with the command: “npm run build”. After the build is complete, can it be started using “npm run start dev”.

The Svelte frontends are set to run on localhost:8080.
If more than one frontend is running, will the instances after the first one, automatically be given a random port, which is easily visible in the terminal of the service.

7. Appendix

Environmental values (.env) for admin & customer backends.

8. References

Links:

https://en.wikipedia.org/wiki/JavaScript
https://nodejs.org/en
https://expressjs.com/
https://svelte.dev/
https://www.mysql.com/
https://graphql.org/
https://www.docker.com/
https://kubernetes.io/
https://swagger.io/
https://github.com/
https://git-scm.com/
https://clickup.com/
https://datatracker.ietf.org/doc/html/rfc5424
https://httpd.apache.org/docs/2.4/logs.html
https://www.altexsoft.com/blog/software-requirements-specification/
https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/
https://www.altexsoft.com/blog/non-functional-requirements/
https://github.com/winstonjs/winston
https://github.com/expressjs/morgan
https://lioncoding.com/logging-in-express-js-using-winston-and-morgan/#
http://tostring.it/2014/06/23/advanced-logging-with-nodejs/
https://datatracker.ietf.org/doc/html/rfc5424#page-11
http://fileformats.archiveteam.org/wiki/Combined_Log_Format
https://httpd.apache.org/docs/2.4/logs.html