Bachelor Thesis

Data management best practices in microservice-based applications

Submitted by
Selma Golos

in partial fulfilment of the requirements for the degree of
Bachelor of Science (BSc)

Vienna, 2020

Degree program code as per student record sheet:	A 033 521
Degree Program:	Computer Science - Data Science
Supervisor:	BSc MSc, Evangelos Ntentos

Contents

Acknowledgements

I wish to express my utmost appreciation to my mentor, BSc Msc Evangelos Ntentos, who has been an integral part of this road. Without his persistent guidance and help, the goal of this thesis would not have been achieved.

I also wish to thank my parents and a dear friend Tamara Khachaturyan, as without their constant support completion of my studies and this project would have been much harder.

Abstract

Microservice-based applications are rising in popularity and one of the crucial aspects for their efficient implementation is their approach to data management. To test the properties of three most commonly implemented data management patterns, namely database-per-service, shared database and event-driven data management, three prototypes of a flight management system were implemented, each implementing one of the patterns. The patterns were developed using the JHipster Framework and tested under different loads for comparison of performance. Each of the pattern implementations came with a certain degree of overhead and therefore the tests were hardly conclusive. However, it is clear that under larger scaling, which is most realistic for such a system, the performance is best with the implementation of database-per-service data management pattern. To conclusively claim that this is a single best option for such a system further, more extensive testing needs to be done. The conclusion most in compliance with the initial research and expectations is that the evaluation of a data management pattern is hardly ever detached from the system it is being implemented in and therefore has to be tailored and chosen carefully to match the system characteristics and needs.

Keywords: Mircoservices, Data Management, Distributed Systems

1 Monolithic vs. Microservice-based Architecture

A traditional approach to developing applications is to develop them as monolithic applications, meaning single unit, centralized applications. Nowadays however, with the raising demand for more complex systems in the technological world this traditional approach does not always satisfy the requirements efficiently. An application built following monolithic architecture pattern is developed as a single, self-sufficient unit, independent of the other applications. In other words, it is designed to perform a system functionality fully on its own, i.e. every step necessary to achieve a certain goal is performed and is dependent on the functionality of that one application. Modules of monolithic applications are extremely tightly coupled due to their centralization. This means that to make any alterations to the system as whole, the entire application needs to be tweaked and improved and as the code base expands this becomes harder. This approach may function very well for simpler applications however, as the complexity of the application grows and requirements change, which is a very common occurrence nowadays, this approach brings more drawbacks. Rising difficulty and challenging scaling of the application, need to redeploy the application after any update is made, limitations put on application's complexity, hardened practice of continuous deployment, slow down in start-up time due to size of the application, unreliability and unavailability, expensive introduction of additions to the technology stack, are only to mention a few. Here is where distributed systems come into play and try and compensate for some of the drawbacks. One of the common implementations approaches of a distributed system is following the microservice-based architecture pattern. Microservice-based architecture follows the principle of single responsibility meaning it groups smaller functionality units that can perform part of a process independently into smaller services which take part in a (distributed) system as a whole. In other words, each service is responsible for a specific task and if necessary, can communicate with other services through APIs for data.[16], Distributed systems are more complex to design and data management does bring additional complexity in decision making during this design process, however they are significantly more scalable, reusable, flexible in technology stack and developer teams distribution and independency as well as simpler and more speedy to develop due to loose coupling and single responsibility principle on the microservice level.

2 Microservices

2.1 Glimpse into the Microservice world

As already hinted the main idea behind the concept of microservices is to break down the system into smaller units and compose the ones that work well together, i.e. the ones that have to be tightly coupled and separate the ones which don't. Each of the components can then be independently developed and maintained which introduces a set of benefits. Even though the microservice architecture does not place any strict boundaries onto the design of the application there are certain characteristics that should be respected to utilize the mentioned architecture. As these characteristics were greatly studied and considered in decision making during the development and data management pattern analysis in this research, they will be explored further. The most obvious characteristic is division into multiple independently deployable components. To utilize this concept to the fullest each microservice should be performing a single function only and should operate independently. Meaning each of the services is developed, maintained and deployed independently. Components are then easier to understand as the code base is smaller, they can be developed by smaller teams and there is no need for extensive communication between the teams which in turn minimizes complexity. Smaller units ensure the capability of easy, fast and extensive testing. They also allow for choosing technology stack so it suits the needs of the service as much as possible, improving performance, ease of development and efficiency of single services which overall contributes to a more efficient system. Independence also allows for perseverance of system's integrity and reliability as the change and redeployment in one service does not affect the rest of the system. The services then communicate with each other through a lightweight communication channel.[10] This communication can be either synchronous or asynchronous, but should be kept simple, and an approach should be chosen effectively. Following request-response model, most commonly implemented using HTTP and following REST[1] principles, ensures that each request receives a corresponding response, either a requested resource or an acknowledgment. However, when scaling microservices it might be the case that not each request requires a response. This is where observer communication comes into play and allows asynchronous and non-blocking communication. Most common implementation of such communication is a message queue such as RabbitMQ[2], Apache Kafka[3], ZeroMQ[4], etc. Both approaches allow for simple transfer of information from point A to point B ("dumb pipes") and elaborate and effective handling of the information at the endpoint of individual microservices ("smart endpoints").[12]

Another logical side effect of microservice architecture and multiple component approach that inevitably leads to a distributed system, is decentralization.

[1] https://restfulapi.net/
[2] https://www.rabbitmq.com/
[3] https://kafka.apache.org/
[4] https://zeromq.org/

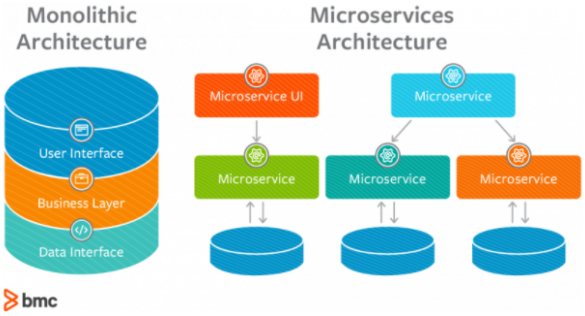

Figure 1: Database difference in monolithic vs. microservice architecture [2]

Services are independent from each other, meaning each of them has their own logic which almost always (if standards were to be followed) leads to having their own data management approaches, making the data management in the system also decentralized. Same as with the technology stack, one size does not necessarily fit all. Data management approach that fits one microservice might not fit all of them, and it is common that each of the microservices employs different strategies in this data handling process i.e. data management is decentralized. In monolithic applications data is managed in a single logical database, whereas a common practice in microservice architecture is logical database per service. This distribution of data leads to minimization of data loss, improved data security to a certain extent and improved data consistency as each service carries only data needed for functioning of that service only. There is no unnecessary data replication and unwanted changing of entire databases due to small changes made for improving a small part of functionality. Both concepts mentioned so far lead to better fault tolerance. Employing independence and single responsibility principle, microservice architecture by default isolates risks of failure i.e. failure points.[7] Microservices are therefore designed to cope well with failure, meaning each microservice failure is handled gracefully and the rest of the system is not put in jeopardy.

Complexity of such a system is naturally higher and requires more careful decision making.Considering these trademarks of microservice architecture ensures higher reliability of the system. Using familiar concepts such as load balancing, timeouts, failover caching, etc. can ensure a development of a reliable microservice-based system. However, none of these principles as already mentioned are not strictly demanded and the decision regarding design is on the development team.

2.2 Drawbacks of Microservice Architecture

Distributed systems and microservice architecture are much talked about as a single-handed solution to all the issues that come with monolithic development approach. However, as suitable as distributed systems can be for certain use

cases and certain systems, they do come with certain drawbacks. The larger the system in question is, the harder it gets to utilize the architecture of microservices to its fullest and develop a system that is significantly more performance effective than it would be if developed using more traditional approaches.

As the system in question scales up, so does its complexity. These systems typically consist of a large number of services, each of them having multiple runtime instances. This is where managing microservices effectively becomes of great importance. Each of the microservices has to be configured, deployed and monitored, health checks need to be performed and a graceful failure needs to be ensured. A service registry and service discovery mechanism also need to be in place to enable successful setup of request-response communication, as the set of runtime instances changes dynamically and so do assigned network allocations.[3] Additionally, as decoupled as microservices may be, they still must be somewhat co-dependent, meaning it is very important to establish effective communication channels and data exchange mechanisms. All of the connections between the modules and databases need to be coordinated and handled carefully. This also demands careful coordination when changes are made to such a highly scaled system.

Another aspect that makes development of a microservice-based system a bit harder is testing. As already pointed out microservices do still depend upon each other to a certain extent, which makes testing that much harder. Each of the services needs to be tested independently, however to test a single microservice, all the services that it depends upon need to be launched, or stubs configured, to simulate their behaviour. As a significant test coverage is necessary to ensure the system is as reliable and as secure is possible this step cannot simply be ignored but the quality assurance teams working on the system must be aware of its complexity, communication paths and details of dependencies between the microservices. Communication types employed might also complicate this process. In the case of asynchronous communication, it might be harder to check just how reliable results are.

Microservices also make it hard to identify where the glitch in the system is. A feature not working in a specific service, might be due to a failure of another service that the feature is dependent upon. These dependencies and effects of failure points on the overall system functionality must also be carefully considered and the issues should be easily identifiable which is almost never easy to ensure.[19] To maximize the chance of easy issue detection, complex monitoring systems need to be in place that would allow for following requests and testing functionality along the communication paths, making in turn the system and development that much more complex.

From the already mentioned challenges of microservice-based architecture it is clear that the definition of boundaries between the microservices might also be challenging, however is essential to minimize the already mentioned drawbacks and to ensure an efficient ground for setting up data management mechanisms. Data management concepts in microservices are essential to understand to follow the development of this thesis, and will therefore be thoroughly covered in the following sections.

3 Data Handling in Microservice Architecture

In contrast to monolithic architecture where the entire application is built over a single database making data consistency that much easier, microservice architecture deals with multiple logical databases. Each microservice deals with certain amount of data native to it, that needs to be kept private and not communicated directly with other microservices to maximize benefits of such an architecture. In other words, it should be a common practice to keep persistent data private to the microservice and accessible to the rest of the system via API in place. [22] However, it is very often a case that microservices need to share or at least communicate the persistent data. In such cases, especially when data manipulation is present, the sharing of data and changes made to it need be carefully coordinated to ensure data consistency across the system. To make this more relatable, we can imagine the case where customer and product management is decomposed into separate microservices, each of them handling customer and product database respectively. The two are very likely to use each other's data frequently, each time a product is purchased the customer data needs to be updated with the new purchase and related to the product in question. They are also very likely to be used together frequently within the rest of the application, during purchase or payment management for example. This where orchestration of data management plays an important role. If handled efficiently the system reaps the benefits of velocity and agility, however if handled otherwise the system becomes less reliable, slower and can easily turn into anti-pattern where it resembles more to a monolithic system i.e. can easily become unnecessarily complex and wrongly implemented. To handle these challenges developer team needs to carefully consider many data management patterns at their disposal and decide on which of them, or which combination of those patterns suits the needs of their system the best and enables the most performance effective yet reliable and secure version of it.

3.1 Database Per Service Pattern

As the name of the pattern suggests, each microservice manages their portion of data i.e. has their own database that is kept private and is directly inaccessible to other services. Communication and data transactions can be realized only through an API. Transactions initiated by a service involve only the data that is part of its own private database. There are several ways that the database can be organized to preserve the core of the pattern without the need to provide a separate database server for each microservice but merely keep the logic separation of the data. [14] In the case of relational databases each service can own a set of tables that can be accessed by that service only, i.e. private-tables-per-service. An alternative is to create a database schema that is unique and closed for that service only i.e. schema-per-service. Lastly the services can be organized so that each of them has their own database server i.e. database-server-per-service. As long as the application and its load do not require a database server for managing high throughput it is safer and cheaper to choose

either private tables or a private schema as an implementation approach since it creates the least overhead while still satisfying the needs of the service and the pattern in place.

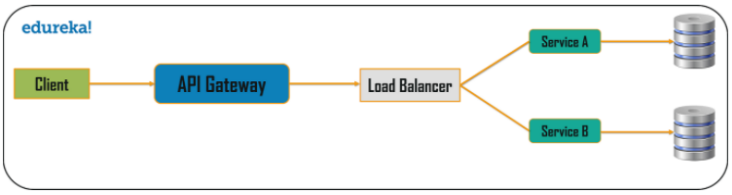

Figure 2: Database-per-service pattern in combination with API Gateway concept [8]

As simple as the pattern sounds it is more often than not much harder to implement efficiently. In most cases, especially when turning an existing monolith application into a microservice-based one, the exact boundaries between the services and data are not clearly defined. Microservices need each other's data for implementing their logic and this can lead to a confusing, spaghetti-like communication. The successful implementation of database-per-service pattern therefore relies heavily on clear boundary separation of microservices and should be thought of when designing microservices, i.e. deciding on microservices or decomposing an already existing system into multiple services. A concept that goes hand in hand with the described approach is API Gateway Design Pattern. Acting as a proxy service which routes requests between the microservices, it enables a much simpler and clearly defined communication protocol, that is necessary for successful implementation of database-per-service pattern. API Gateway serves as an entry point for all services and is able to route and then aggregate results back to the original service.[8] Used in combination with load balancer and service discovery the load of the requests is handled and routed correctly between the services. This then enables a simple stateless communication between the microservices that is imminent when implementing the data management pattern in question.

If implemented properly, a single database per service means even less coupling between the services. This in turn opens up many possibilities for the developer teams. They can choose different database solutions for different microservices based on the needs, i.e. if a service has a different set of storage requirements, they are easily manageable as these decisions are no longer dependent on the entire system but are relative only to the service in question. Scaling of a service then becomes much easier and independent from the rest of the system. This pattern also supports the reduction of data duplication as each service uses and writes only what it needs. There is no need to duplicate the data as all the data that is necessary for the logic implementation and belongs to another service is requested through an API. This in turn also reduces the inconsistency of the data as writes to the database cannot be done by no one except the native microservice, i.e. single writes per logical unit of data.

When implemented in larger applications, or the ones with large scaling predictions, the exchange of data becomes unnecessarily hardened. With each transaction API needs to fetch and forward the data, interacting with numerous databases from numerous services. For a complex business use case, this manner of communicating creates huge overhead and if avoidable it should be reconsidered. Implementation with queries that require join of data from multiple databases becomes challenging as well as the management of numerous databases.[14]

3.2 Shared Database Pattern

Dealing with complexity that arises from implementation of Database-per-service data management pattern can become quite hard, especially if not starting the development of a new application but switching from a monolith one to a microservice-based one. Decomposing an application to services and denormalization is often not that easy. Shared database pattern is a quick and simple solution to these challenges. As the name suggests multiple microservices share the same database which is not an ideal solution but is a working one. It enables the developer team to utilize an already existing part of architecture and work in already familiar manner.

To preserve the consistency of data in case of this implementation the transactions should be ACID[5] compliant standing for Atomic, Consistent, Isolated and Durable. A transaction especially in the context of microservice based application over a shared database, must be completed in its entirety or not completed at all. Either the transaction is successful or not which complies with the second principle of keeping transactions and the database state before and after the transaction consistent. Transactions are isolated from each other and executed independently from one another, meaning the data can be queried independently regardless of the change being made to it in parallel. The user will see the results of the query before the change and if requested again will see the change made. Lastly all the transactions must be able to be recovered from, i.e. they need to be kept in longs or durable in any matter that enables full recovery if necessary.

Even though compliance with ACID properties keeps the shared database of microservices as reliable as possible, the pattern itself is far from an ideal solution for data management in a microservice-based architecture. It acts as an anti-pattern in a way, as it negates the core purpose of division of the system into services, loose coupling. Services sharing the database become again highly dependent on each other and the decoupling between them is brought to a minimum. Developer teams need to communicate all the changes made to the database schema as the change now affects the system in its entirety. Execution of this pattern should be done carefully as the load of multiple microservices, upon runtime with number of instances each, using the same database can lead to overloading and runtime conflicts when trying to access the same database

[5] https://database.guide/what-is-acid-in-databases/

resource. Scaling of the services in this case also becomes significantly more complex.

However, aside from the number of disadvantages it carries, shared database pattern does provide an easy and a working solution to challenges and complexities of the database per service pattern, which sometimes all that is necessary. If kept to a small number of services and not an entire system it could be useful.[see Figure 3]

3.3 Event Driven Data Management Pattern

So far mentioned patterns focus on synchronous communication between the services, meaning each request must wait for a response i.e. there is a degree of blocking and waiting for a response. In cases like communication between the gateway and UI or gateway and the rest of the system, this synchronicity is more than welcomed as it makes sense to have a request-response communication. In other cases, such as inter-service communication, the speed of communicating can be improved by switching to asynchronous communication patterns. Other use case important to consider before delving into the pattern in question, is an already mentioned one, when certain transactions include multiple services. As already discussed, this can be quite an issue if following database-per-service data management approach, and a solution for it was to use a shared database. However, as mentioned, as quick and effective as this solution is, it diminished the most benefits that the microservice-based architecture brings and should be avoided if an alternative exists. An alternative that would cover both use cases mentioned, benefits of inter-service asynchronous communication and effective transactions spanning multiple services, is event-driven data management pattern.

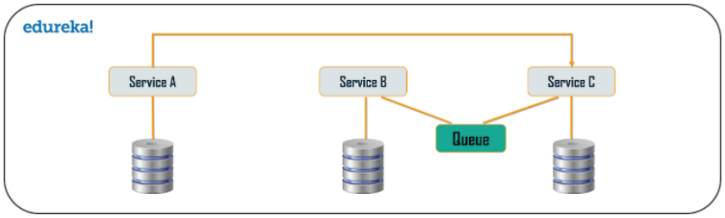

Figure 3: An abstract representation of event-driven data management, where microservices communicate via events stored in a queue asynchronously, whereas the communication with the rest of the system is kept synchronous. [8]

An event in the context of a system would include any important change made to that system, such as creating, updating or even deleting a resource in the system. Services then communicate via these events. A microservice publishes an event if something worthy happens, an update being made to its database i.e. resources for example. These events are not published directly to other services, but to a message broker (message queue) or a publish/subscribe

model stream where they are persisted for a certain amount of time. Microservices that are interested in these events subscribe for them and can handle the data shared within them as pleased, for example to update their own resources which than leads to more event publishing.[6] Services that publish events fall into category of producers and the ones using those events fall into category of consumers. The core responsibility of the producer is to inform about an event through an event notification. Events as already mentioned are persisted in a way and are available for grabs and processing in the future. Once this event source is established, an event stream in the form of a message broker for example, stores the number of events in the same order they were published in the form of a queue and allows parallel processing by several consumers.[6] This differentiation between services, producers vs. consumers, might be useful when switching to an event-driven data management pattern.

As the other patterns, event-driven data management has several benefits for the developer team but also comes with several drawbacks. The most commonly mentioned benefit is allowing for business transactions across multiple services without the need to use distributed transactions and while promising eventual consistency. Consuming an event triggers the data update and potentially initiation of another event, creating a coherent flow of change which enables completion of transaction across services and eventually ensures data consistency. Here it is important to note that these transactions can be implemented efficiently if and only if ensuring that each service is capable of atomically updating its data and publishing an event about the update and ensuring that the message broker delivers the published event at least once.[6] Another clear benefit is performance enhancement as a consequence of asynchronous messaging in place and preservation of loose coupling between the services. The services do not need to know each other's data or any specifics for that matter, they just need to know to which topics of interest they publish to i.e. subscribe for. In turn since services are loosely coupled, scaling becomes much easier. Keeping events in a queue for a certain period of time enables for recovery options. Similarly, to version control systems we are all so used to, this property allows for recovering the lost work and minimizing and preventing data loss by going back to previous events and reapplying them in a way.

However, with such large buzzwords for benefits it is clear that certain drawbacks are dragged along. The system implementing event-driven data management comes with a much more complex programming model. Transactions are no longer ACID and the system must be designed in a way so it is capable to compensate for this lacking property and deal with possible data inconsistency. If read from a view that is not yet updated as the event handling has not yet been dealt with, inconsistencies are visible and this should be prevented. Therefore, in order to be as reliable as possible the system must ensure that the database is updated and an event published atomically and for this it cannot use traditional approaches that utilize distributed transactions spanning both database and the message broker.[15] In this case patterns such as Event

[6] https://examples.javacodegeeks.com/event-driven-data-management-for-microservices/

sourcing[7], Database triggers[8], Transactional Outbox[9], etc. can be implemented, which in turn again increases complexity of the system.

3.4 Data Management Considerations to Remember

The most benefits from the microservice-based architecture are gained when the application is in need of large scaling. Applications of a high scale naturally deal with vast amounts of data and this data needs to be managed efficiently if the benefits of microservices are to be kept and utilized. Distributed systems most often mean distributed data i.e. decentralized data management as already discussed. This is usually achieved by allowing each microservice to manage its own data and this where data integrity and data consistency should be paid special attention. There is no single correct approach to manage the data in microservice-based systems but there are several guidelines to keep in mind when deciding on a data pattern to implement. Same guidelines have been considered during the case study examination for the purpose of the thesis end goal.

Points in the system where a strong consistency is a must should be clearly identified and transactions made around it should closely follow ACID principle to achieve that consistency. On the other hand, point in the system where data consistency is slightly more lenient can allow striving for eventual consistency guaranteed by patterns such as event-driven data management. Points which require strong consistency should follow a request-response pattern and strive for synchronicity. Transactions triggered by this process and performed in the background can again strive for eventual consistency. This approach would follow database-per-service pattern with the part of the system following API Gateway Pattern for example and a part following asynchronous communication using a message broker for example. Either way the business changes that transform the data need to be communicated via standardized messages, be it using APIs or message queues.[17]

System as a whole should strive for storing only the data that is necessary, meaning each service should store only the data that it crucial for logic implementation. Data redundancy and copying across services should be minimized, which contradicts the implementation of a shared database pattern. Coherence and loose coupling of services should be maintained instead. If two services need to share too much data, perhaps their design should be reconsidered and functionality should be decomposed into services in a different manner. However, if that change in design would trigger too much overhead and too much complexity regarding the development, a partial implementation (for the microservices in question only) of shared database might be an easy, working fix.

Consider if the functionality of different microservices has different data management requirements and if the case in question could benefit from polyglot persistence of data i.e. different database implementations for different services.

[7] https://microservices.io/patterns/data/event-sourcing.html
[8] https://microservices.io/patterns/data/database-triggers.html
[9] https://microservices.io/patterns/data/transactional-outbox.html

If yes, to which extent and is it worth the complexity? Storage demands are crucial here. The service needs to take into consideration the amount of data that will need to be stored, the speed at which it will be receiving the data and in which way will it need to be processed. How secure does this process need to be? These are the questions that should be considered when deciding on an appropriate data repository and on whether that repository is suited and necessary for all microservices.

How much complexity should be risked for the benefit of performance effective solution. In a highly scaled application, especially if that application deals with constant flow of data such as social networking applications or applications dealing with processing of IoT sensor data, event-driven data management can be extremely useful and can speed up the performance to a great extent. However, in the applications that deal with simple user input and distinct data entries that need to be processed, would it be worth the complexity to gain performance improvement, if any, by following the same pattern.

These are some of the aspects that were studied throughout the case study to follow. All of the mentioned data management patterns have something to bring to the table and based on the system requirements, functionality and scale an informed decision on which pattern to follow can be made. This is exactly what will be tested throughout the case study with the hope of seeing which data management pattern comes with which extent of trade of between complexity of the solution and the performance gain.

4 Case study

In order to evaluate the most common data management patterns for microservice-based applications a case study has been conducted. The core of this case study lies in testing the performance of the three already introduced data management patterns: database-per-service, shared database and event-driven data management in an environment that mimics the real-life scaled application. To this end three prototypes of a flight management system application have been developed, each implementing one of the three patterns. Test cases have been extracted from use cases of the application and been tested against each other to compare the performance of the prototypes. Based on the test results the three patterns have been evaluated in the context of the developed application and discussed further in the broader sense.

4.1 Technology Stack

The technology used for the application development is: JHipster Framework[10] for more automatized implementation process, Java 11, Spring Boot Framework[11], Angular Framework[12], Maven[13], SQL, Apache Kafka[14], Docker[15], Gatling Framework[16] for performance testing.

4.2 FMS: Flight Management System

The goal of the case study was to implement a microservice-based application using JHipster Framework whose functionality lies in flight management. The application should serve a passenger primarily. Therefore, the main functionality of the application lies in providing the passenger with a list of all available flights and enabling the booking of a specific flight. Booking process includes adding luggage information, checking flight information and receiving a confirmation after a successful booking upon successful payment. The system differentiates between admin and passenger type users. Admin has access to all of the application's functionality and data and is able to register new flights, cancel the existing ones and manage registered user accounts. Passengers on the other hand are able to register i.e. login into their account and browse and book flights. They can also view their booking history, payment history, received invoices and make changes to the booking, e.g. cancel it. As the application is microservice-based, its functionality was decomposed into six microservices and an API gateway including passenger microservice, flight microservice, booking microservice, luggage microservice, payment microservice, and notification microservice.

[10] https://www.jhipster.tech/
[11] https://spring.io/projects/spring-boot
[12] https://angular.io/
[13] https://maven.apache.org/
[14] https://kafka.apache.org/
[15] https://www.docker.com/
[16] https://gatling.io/

4.3 Detailed Design and Concept

The final architecture and software design of the Flight Management System application has been thoroughly thought through taking into consideration passenger needs and the patterns for data management. The system was modeled using UML[17] modelling kit and taking advantage of its behavioral and structural diagrams. Use case, activity and sequence diagrams were used in the initial phases of design to extract concrete behavioral requirements of the application from the general idea started with. Component and Class diagrams were then used to plan the architecture of the system (crude and detailed) that would satisfy the set-out requirements from the previous planning steps. Firstly, user stories were thought through so that the system can be checked against mandatory functional requirements throughout the development process. Following user stories were deemed as mandatory to fulfill and are displayed in the use case diagram (see Figure 4):

1. As a passenger I would like to be able to login into the application's account with my credentials or if I do not yet have an account, I would like to be able to register for.
2. As a passenger I would like to have an overview of available flights so I can select the one to book.
3. As a passenger I would like to be able to book a flight I find matches my wishes. To do this I need:
    - to browse through available flights (see user story no. 2)
    - to select a flight that matches my wishes for departure date, flight type, flight price, and flight source and destination
    - to add the information about the luggage I plan to take on the flight
    - to pay for the chosen flight in order to complete the booking
    - get the invoice after completion of the flight reservation
4. As a passenger I would like to be able to look through my bookings on my account.
5. As a passenger I would to be able to cancel or update the booking information on my account.
6. As an admin I need to be able to login into my account.
7. As an admin I need to be able to update the flight schedule and/or other flight related information (including addition of a new flight)

[17] https://www.uml.org/

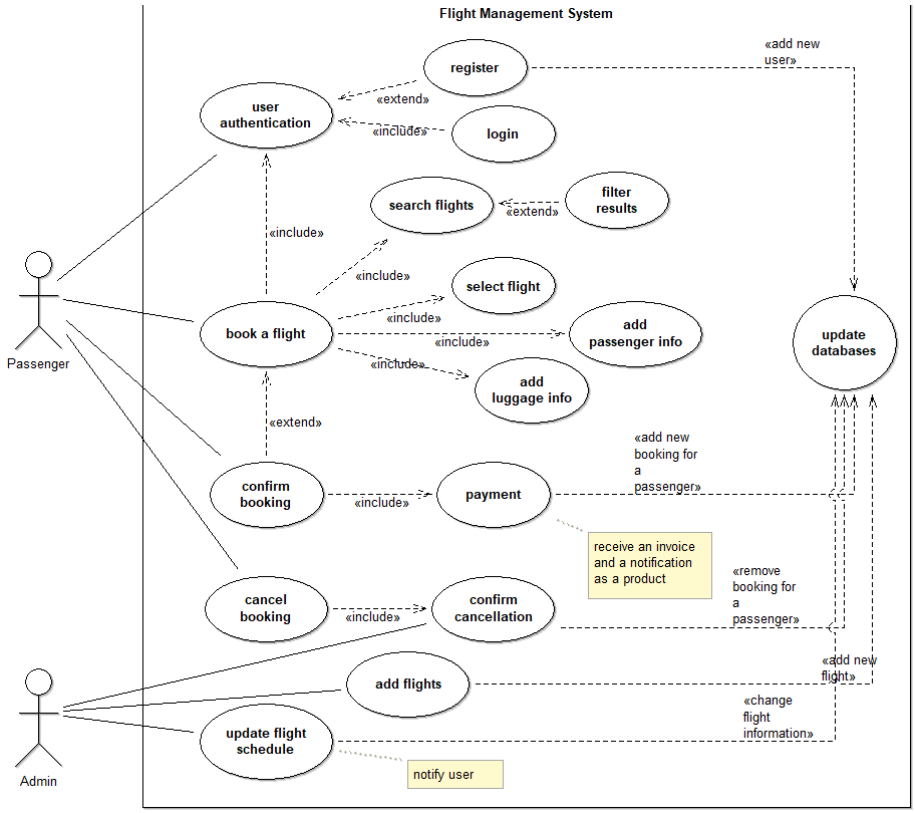

Figure 4: Functionality of Flight Management Application displayed through a series of use cases.

The component diagram below (see Figure 5) simply displays the already mentioned rough architecture of the system. The flight management system is as mentioned comprised of six microservices and each of the microservices is registered at JHipster registry upon starting. An API Gateway is provided as well which serves as an aggregator of certain application processes and intermediary between the system and User Interface it provides the passenger with. Both microservices and the gateway have pre-configured ports displayed in the component diagram.

To understand the showcased diagrams better and to understand how the services handle and share their data i.e. communicate within the system it is crucial to understand what is the scope of each of the microservices. Flight microservice is responsible storing and managing available flights. It allows CRUD operations over its database to the admin and read operation to the passenger. For each flight departure data, origin and destination, flight type and price are recorded along with the flight number. Each flight is related to an airline it is a part of, airport, boarding gate within the airport and a plane

that the passengers board. Passenger microservice allows for management of passengers, checking for authentication credentials, and relation of the passenger to the rest of the system. Each passenger during the booking process is able to add luggage they will take on board, and this luggage is managed by the luggage microservice. Tags for luggage recognition as well as certain relevant stats such as weight and category are recorded and managed. Luggage is therefore upon adding during the booking request, related to the booking that the passenger initiates for a specific flight. Booking process also demands payment for successful completion. All payments are handled by the payment service which checks for the credit card information of the passenger that initiated the booking and manages the payment via CRUD operations performed over the native database. Payment service also generates and delivers an invoice upon successful payment that the passenger can then view in their account. Notification microservice acts as a silent observer of important events and notifies the passenger on important events such as successful booking confirmation, booking cancellation and flight updates i.e. flight delay or cancellation. Having in mind this structure of the system makes it easier to understand how the services should communicate with each other in each of the patterns that underlie the evaluation of data management in microservices in this case study.

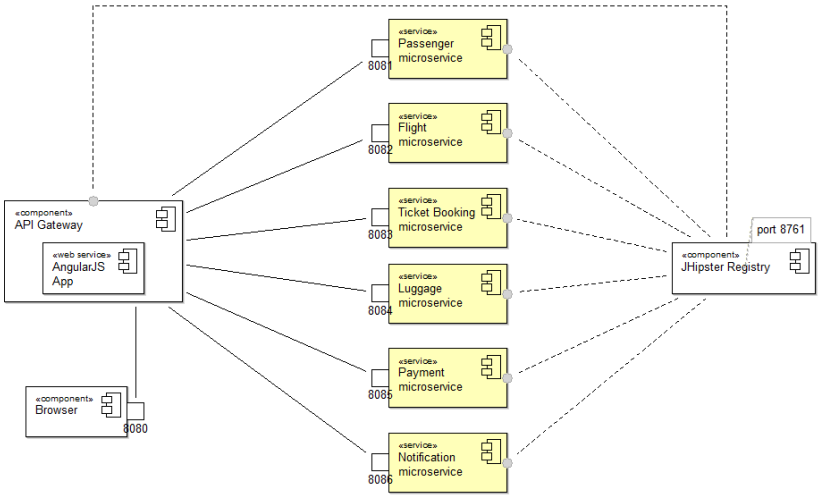

Figure 5: More abstract view of Flight Management Application showcased through a component diagram.

A slightly more detailed insight into the communication within the system can be seen in sequence diagrams showcased below. User Login (Figure 6), Booking of a flight (Figure 7).

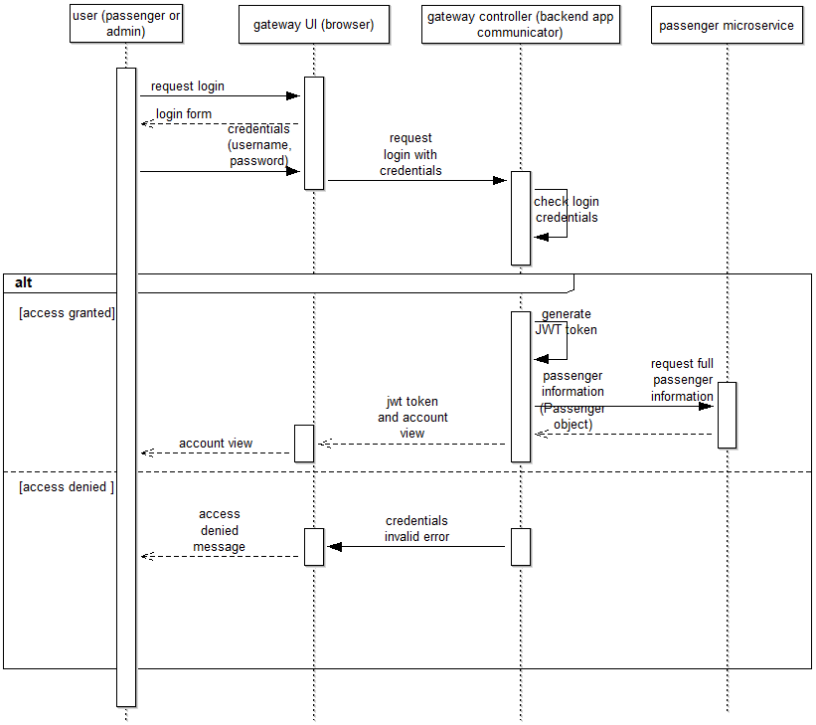

Figure 6: Communication flow for user login use case.

4.4 Related Work

To be able to implement the already mentioned three prototypes of flight management system and to be able to evaluate and discuss the performance of each of them i.e. each of the three data management patterns, a detailed research and analysis of already existing information on the topic was conducted. Most frequently used resources were already mentioned throughout the paper and have helped gain a deeper understanding of microservice-architecture demands[7] [3] [17] but here they will be focused on data management patterns, the right way of measuring their performance and current best practices when it comes to data management in microservice-based architecture. Another focus will be more pattern implementation topics, such as event-driven microservice communication with Apache Kafka or the role of API Gateway Pattern in implementation of data management patterns.

Decentralized data management as a concept that supports the natural characteristics of microservices and main contributor to polyglot persistence has been introduced as early as the term microservice.[7] The cleanest approach to achieving full decentralization and keeping the loose coupling of services is to separate single logical database the monolith is used to into logical database

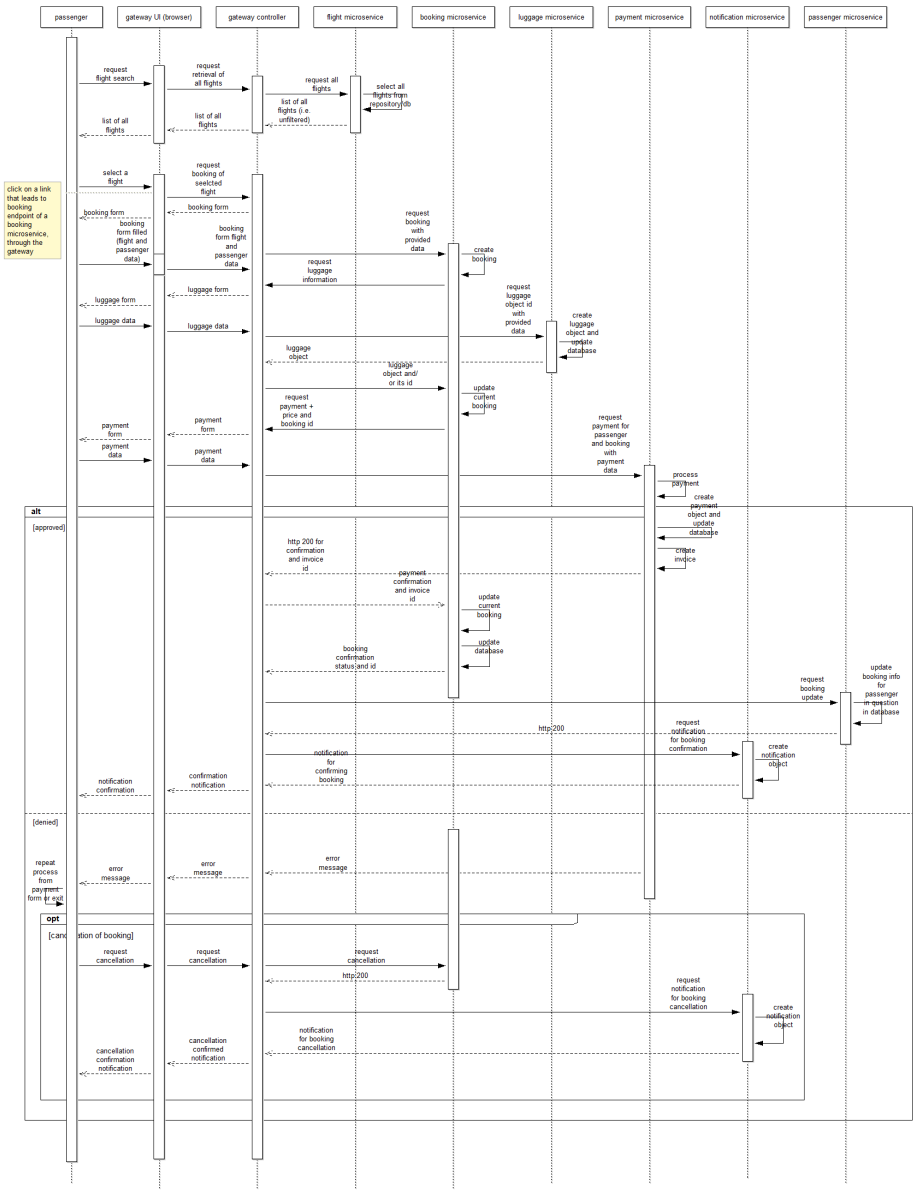

Figure 7: Communication flow for booking of a flight i.e. main use case.

per service, whether it be a table per service, schema per service or if necessary, a database server per service.[13] To avoid direct client to microservice communication and avoid the complexity that comes with it, this and other patterns often come along with the API Gateway pattern, i.e. using an API gateway, a server that acts as a single-entry point into the system. It encapsu-

lates the internals of the system and provides an API as an access to systems functionality, along with provision of load balancing, monitoring, authentication, caching etc.[5] Once the complexity of such a system due to high scaling comes with too much overhead for the development team it is a common solution to introduce a shared database pattern to a certain extent, as it allows for an easy and working fix in terms of simple data sharing between the microservices, however it is a bad practice long term as it diminishes the purpose of microservice decomposition.[18] [21] When optimizing highly scaled applications that depend on a large amount or a constant flow of data, many have found they can benefit greatly from an event-driven data management, where each change to system will be followed by an event publishing to a message queue where it is ready for consumption when necessary. This asynchronous communication allows for performance benefits of non-blocking communication during a process execution.[6] [4] [13] To support such an implementation many utilize a reliable, open source project maintained under the Apache Foundation, event streaming platform that is able to sustain operations on a massive scale, Apache Kafka. It allows for data replication across cluster of services which aids the fault tolerance policy as it is a distributed platform. It allows for use of traditional queuing and publish-subscribe model yet remaining scalable when the messages are consumed by multiple points and ensuring receipt of messages in the order they were published.[11] [20] More detailed overview of characteristics that make it stand out in the market can be picked up from the mentioned resources, however the mentioned qualities concern this case study the most. However, the increase in complexity due to event driven architecture is worth considering if this approach is truly necessary.

Deciding on a data management approach to follow, as seen throughout the research, is directly related to the functionality requirements and the needs of the application in question. Type of data dealt with, regulations and security needs for that data and business process in question need to be considered. Probability of high scaling of the system, storage requirements for data volume as well as importance of data consistency at all times play an important role in choosing the right data management approach and the right choice of databases.[17] [1] The way to quantify the decision-making process would be to consider the read and write performance, resource and provisioning efficiency and latency. [9] Metrics such as number of operations per second, how fast can queries be ran and results retrieved or number of write operations per second can be a good way of measuring the capabilities of the chosen database, organization of the data and therefore the performance of the system under a certain data management mechanism. On the other hand, having to provide instant user experience requires a low latency database and is more prone to benefiting with patterns such as database-per-service. The system also needs to be agile and able to sustain the needs of rapid development, testing and production as well as high scaling meaning the data needs to be able to scale on demand, i.e. has to be resource efficient and creation of numerous instances per second on demand needs to be efficient.[9]

Many resources cover the general best practices when it comes to develop-

ment of microservice-based systems and many discuss the available choices when it comes to data management however, what this case study can contribute is an experiment-like research on the topic. It compares the three most common data management strategies on a simple flight management system, and places the focus on performance of each of the patterns in same use cases under the same parameters, which allows for a more focused and direct comparison between them.

4.5 FMS: Prototype I (click for source code)

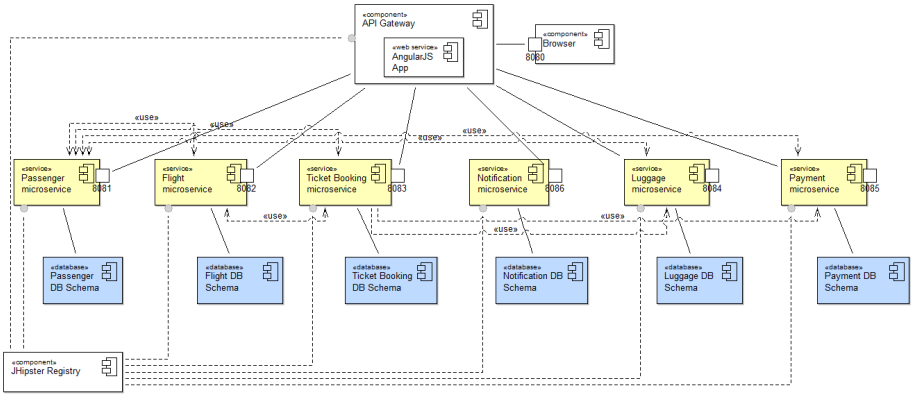

Figure 8: A more abstract view of the architecture implementing database-per-service pattern of the prototype I.

The idea behind implementation of the flight management system application with a database-per-service data management pattern is that each microservice's persistent data is private to that service. The pattern is achieved by creating either a table or a schema per microservice. The communication between microservices is API based and the data from the databases is never accessed directly by another microservice. Likewise, the transactions of a service can include only its own database. Each of the services as pointed out has a database schema of its own to satisfy the architecture suggested by the pattern in question. SQL relational databases were created for each of the microservices and were generated based on the definition laid out in the JDL files (available in the source code repository). Upon JHipster application generation the entities representing tables of the database were generated based on these files.

The application was developed with the use of JHipster Framework, and has followed the characteristics of the FMS application already described, i.e. division into six microservices and an API Gateway. API Gateway serves as a User Interface and a single-entry point to the rest of the system. All of the client requests are forwarded to the microservices to perform a certain functionality through the gateway. To dissect the system's implementation approaches to

following functionalities were described in more detail: User authentication, Flight search and adding and booking of a flight.

Regarding the user authentication the idea was to utilize already generated authentication and authorization system by JHipster. JHipster uses JWT authentication and Spring Security to differentiate between two types of JHI Users: Admin and User, where Admin has authorization to manage the users (CRUD operations over User database). The idea was to refactor this implementation to make the differentiation between the Admin and Passengers of the flight management system. To simplify the process and not to refactor larger portions of the auto generated code and cause overhead in execution, JHI user was treated equivalent as a passenger, and admin as an admin of flight management system. **Admin can login into his account regularly with username: admin and password: admin**, and has the authorization over user (i.e. passenger) management. The passenger management by admin, and updates of the database in passenger microservice (through gateway) were simulated in a way that each time an admin creates/deletes/updates/views a user in the UI, the data is either checked for or added/updated in passenger database (by calling an endpoint of the passenger microservice by the gateway). **A user can login into their account using username: user, and password: user.** They can view their account and upon each login the passenger database is checked for the same account, if the one does not exist an error is logged. If the user registers the passenger database is updated by, again, calling a corresponding endpoint of the passenger microservice. This way the functionality of user authentication is kept mainly on the side of the passenger microservice but the already generated template for user authentication is fully preserved.

From the user perspective flights can be browsed through (retrieved from the database in Flight Microservice). Admin has the rights to adding new flights, and deleting them, same as all parts necessary for a flight to be registered properly in flight microservice (flight handling, airline, plane, airport management). User however does not have the rights to deal with flight handlings, airlines, planes or creating, deleting and editing the information about a flight. The flight can be viewed in more details the user can request booking of a flight from the overview of all flights by clicking on an 'Book Me!' option. This opens a form for the new booking creation, where the flight number and price are filled in automatically. The user is upon saving of a booking redirected to a page where he/she can add luggage wishes. The data that is already familiar in the luggage form is also filled in automatically and when filled completely and confirmed, the luggage is added and user is redirected to the payment/confirmation section. The payment form is opened and the user needs to confirm the data. The payment process is merely simulated as no real credit card information is checked, not the real payment via external services is completed. Once the payment is confirmed an invoice is created and user needs to confirm the auto filled data in the invoice form i.e. double check their booking. After this confirmation the booking process is finalized and user is notified with a pop-up notification on the screen. The notification the user sees comes from the request for a 'booking confirmed' type of notification from the Notification Microservice. Same hap-

pens if the user is notified about a booking cancellation, flight cancellation or update.

4.6 FMS: Prototype II (click for source code)

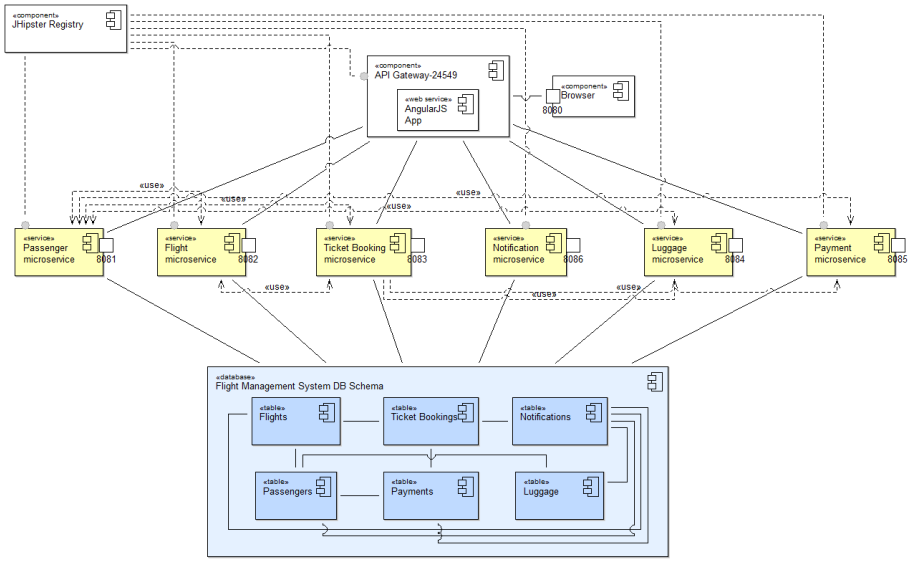

Figure 9: A more abstract view of the architecture implementing shared database pattern of the prototype II.

The second prototype is meant to showcase the properties of the shared database data management pattern. Therefore, the functionality of the system and the way it is implemented has been kept the same as in the first prototype. API Gateway still functions as an intermediary between the client and the rest of the system. The user interface provided remains the same as do the access endpoints into the system. Meaning REST based endpoints provided by each of the services for manipulation of their resources (database entries) remain the same. In contrast to the first prototype however, all of the services' independent databases have been removed and one common database has been created. The database schema that is shared between the services can be seen in JDL file "shareddbconcept.jh" in the fms-prototype-2 folder in the repository provided via link above. The idea was that the database is created on the booking microservice and then accessed by all the microservices. The changes made to the system, needed to achieve this type of data sharing were mainly in:

- flight-management-system-prototype-2-service.yml
- flight-management-system-prototype-2-service.yml

- flight-management-system-prototype-2-serviceh2.server.properties
- flight-management-system-prototype-2-service.yml
- flight-management-system-prototype-2-service-testcontainers.yml

The rest of the code was mostly kept as it is. Notifications service remained untouched as well as the gateway. The services were left only with entities they are in charge of even though they all read/write to the same database. So, for example in the booking service only the management of Booking Entity can be seen even though this service holds the entire database. In the Flight service there is the management of Flight, Flight Handling, Airport, Airline and Plane entities as before, and all the management results in the changes of the database held by booking service (so a shared one) but the other entities remain untouched by flight service.

4.7 FMS: Prototype III (click for source code)

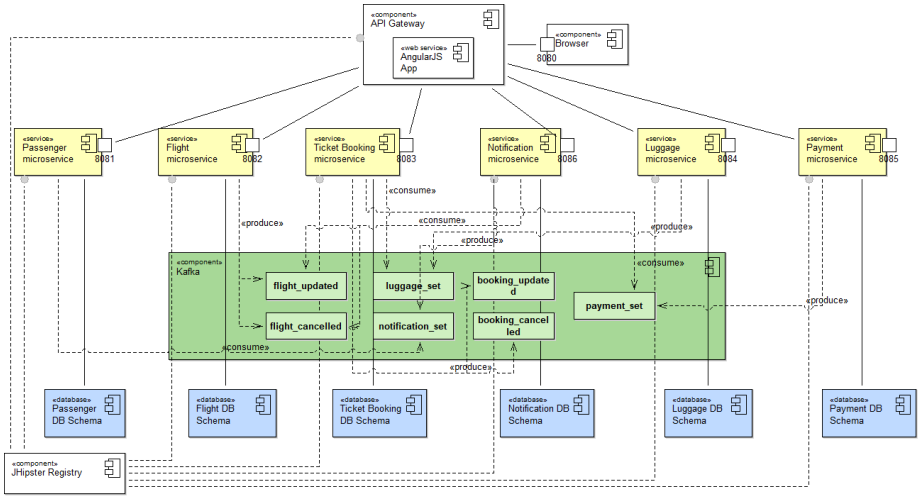

Figure 10: A more abstract view of the architecture implementing event-based communication in prototype III.

The third prototype is showcasing the event driven data management pattern is implemented using asynchronous messaging property offered when creating a microservice application with JHipster. Each of the microservices has its own database and communication between them is organized asynchronously. When a major event happens, i.e. any type of an update to one of the databases, if relevant, the event is published to the broker with a corresponding topic. If interested microservices can subscribe for the topic. In this manner the communication within the system, i.e. between the microservices, is non-blocking and can contribute to better scaling of the application. Alongside of microservices an

API gateway is implemented as it was the case in first two prototypes, and the communication between the API and the system is kept synchronous and REST API based. The API as before provides the user with an interface and is the intermediary between the system and the user. The user therefore needs feedback, i.e. regular response to their actions from the gateway, meaning synchronicity that request-response communication provides. The rest of the implementation was kept as it is and the change was introduced only when determining which of the microservices will act as producers, which as consumers and which as both. Producers would use their Kafka connection to publish an event each time their database is affected, and consumers, depending on interest, would consume the events they receive from Kafka based on their topics of interest. The consumed messages are used to either check validity of entities or to trigger the notification creation for certain events. The detailed overview of the topics in play and consumer-producer role distribution can be seen in the component diagram provided. See (Figure 10)

5 Evaluation

5.1 Test Cases

The following test cases show the most realistic use of the flight management system and include both admin and passenger use cases and grow in demand from the first case to the last one which uses all of the microservices.

1. User i.e. passenger login with username: user and password: user and browsing of flights.
2. Admin login with username: admin and password: admin, and creation of the flight.
3. Admin login with username: admin and password: admin, flight creation and deletion (cancellation).
4. User i.e. passenger login with username: user and password: user and booking of a single flight.

5.2 Testing Strategy

To be able to compare and contrast the three prototypes i.e. three data management patterns in the context of the case study described, their performance was tested against the four test cases laid out. For this purpose, the Gatling Framework was used, an open-source load testing framework. Gatling enables the creation of scenarios that are based on how the system is meant to be used and navigated by a real user. The scenarios created in the context of flight management system were created based on the test cases described. The scenarios are then converted into Scala-based Gatling simulations[18] using the Gatling recorder[19]. The simulations can be tailored as necessary and, in this case, have been to simulate different number of active users of the system. The admin related test cases, where admin logs into the account creates and/or deletes a flight, have been tested across prototypes for only one active user as the admin role should be kept unique. For the two test cases where the passenger logs into their account, browses flights and finally books a flight, the number of active users was alternated between 1, 100, 1000 users at once and finally a variation where the simulation starts with 10 active users, for the next 10 seconds adds 10 users per second, and then during 60 seconds ramps up the a 1000 new users. The role of this was to test how the response time and error rates change with the change in load to the application.

The prototypes were tested on a local PC with following stats. Operating System: Windows 10, Hardware: 2 Cores, 12GB RAM, 256GB SSD Disk.

[18] https://gatling.io/docs/current/general/simulation$_structure$/
[19] https://gatling.io/docs/current/http/recorder/

5.3 Result Discussion

5.3.1 Test Case I: User login and flight browsing

Prototype No.	No. of failed requests	RT under 800ms	RT under 1200ms	RT over 1200ms	Min. RT (ms)	Max. RT (ms)	Mean RT (ms)
1 active user (11 requests)							
I	0	11	0	0	11	518	95
II	0	11	0	0	4	171	37
III	0	9	0	2	18	1675	381
100 active users (1100 requests)							
I	0	931	5	164	3	5062	592
II	0	921	10	169	3	6037	628
III	0	849	24	227	2	6419	792
1000 active users (11000 requests)							
I	0	1938	892	8170	5	50231	6282
II	0	1968	515	8517	4	55632	7357
III	0	1852	471	8677	8	53326	6975

Table 1: Gatling response time (RT) measurement overview

User login and flight browsing test case involves the checking of credentials on the gateway side to finalize the user login, the interaction with the flight microservice, i.e. one request to a microservice to fetch all the flight data from the database and one request to view details of a flight, and interaction with the bookings microservice to view the current user bookings. When going through this scenario and tracking the information exchange, it amounts to the total of 11 requests. The scenario is still kept simple and was expected to perform without failure in requests, even when testing with 1000 active users at once. From the overview of the response times (see Table 1) it can be seen that results on all three prototypes are quite similar. The requests that had the longest response times overall are the ones providing the login data of the user and the once fetching all the flights from the flight microservice as it was to be expected. For a single user it is also expected that readings are quicker when done from a common database, i.e. following the second prototype, and that the overhead of the third prototype's complexity slows down the system in general. However, one active user is rarely a credible representation of the system's performance. When scaling to 100 and 1000 users it can be seen that the first prototype takes the lead, whereas the third prototype gets closer to it the more it scales (when considering mean response time) i.e. the more the system scales the smaller the performance gap is in database-per-service and event-driven data management implementation whereas shared database implementation scales worse.

One additional setup has been tested, in which the system starts with 10 active users. Then over the next 10 seconds it gets 10 new active users per second, and finally during next 60 seconds it ramps up to 1000 new users. The

intention behind such a setup was to simulate the most realistic use of the system yet keep it under a relatively high load. This amounts to 1110 active users throughout the simulation and to 12210 requests made to the system. As the test case is still simple failed requests were not expected, and there were none. The response time overview can be seen in the figure below (Figure 11), percentile-wise with the focus on minimum and maximum response time.

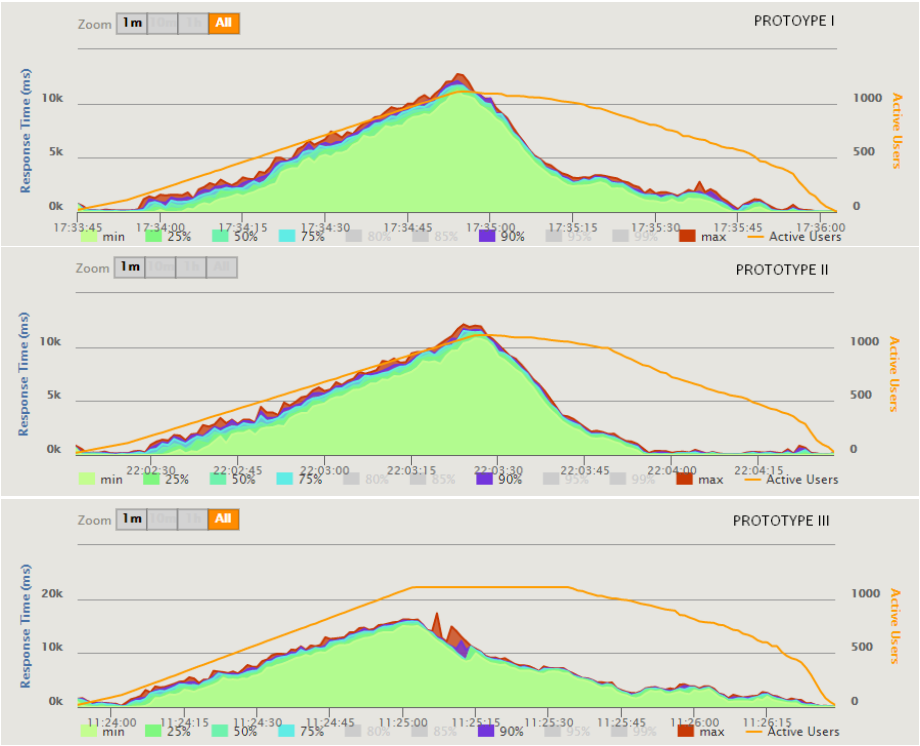

Figure 11: Response Time percentiles over time

If we take a look at the three graphs displaying response time percentiles over the course of the simulation, it can be seen that the shared database prototype is the fastest, database-per-service is a close second, whereas the event-driven implementation using Kafka is the slowest. It is very possible that if the tests were to be repeated the results of first two prototypes would vary and still be very similar, which is to be expected for a simpler test case. The overhead that decoupling of data and services, that the database-per-service pattern brings to the table, can likely be the cause of a slightly slower response time in comparison to communicating via shared database. Implementation of third prototype with Kafka still comes with the greatest overhead as the use of Kafka itself is very heavy and the application under this test is neither tested with a complicated enough test case, nor is it scaled enough to benefit from such an implementation.

5.3.2 Test Case II and III: Admin login and flight creation (II) and cancellation (III)

Prototype No.	No. of failed requests	RT under 800ms	RT under 1200ms	RT over 1200ms	Min. RT (ms)	Max. RT (ms)	Mean RT (ms)
Test Case II							
I	0	22	0	0	5	232	61
II	0	22	0	0	10	373	89
III	0	20	2	0	10	1184	207
Test Case III							
I	0	28	0	0	6	159	48
II	0	28	0	0	5	143	48
III	0	28	0	0	5	139	50

Table 2: Gatling response time (RT) measurement overview

Admin login and flight creation test case involves the checking of credentials on the gateway side and interaction with the flight microservice, i.e. one request to a microservice and a create operation over the database. When going through this scenario and tracking the information exchanged with the FMS, this functionality amounts to 22 requests. As the test case is fairly simple and performance was measured for a single user only, the hypothesis was made that all requests should pass error-free and within 800 milliseconds. Out of the requests made to the system, in general across prototypes the longest response time had the ones that were fetching .js and .css files, and the image to display on the main page of the system, which is to be expected. However, the requests to the flight microservice database to fetch all the flights, had the longest response time in third prototype which is supposedly due to overhead that the application's structure carries with. The second longest request was admin login, and in general it took slightly longer to save a newly created flight. By simple use of asynchronous messaging the third prototype's structure rises in complexity which can lead to longer reactions when testing unscaled application. However, if the tests were to be repeated several times, the response times would vary and probably equate among the prototypes more often than not, therefore it is to be concluded that measuring such a simple test case against one active user is merely a good test of the apps functionality and general response time, but is hardly telling much about the properties of data management patterns.

Third test case in which admin logs into their account, creates a flight, and deletes it is slightly more demanding as it involves a few more operations. It mimics the second test case and in addition to it has one more request to a flight microservice and an additional delete operation over the database. It also interacts with the passenger microservice as it creates a notification about the flight cancellation that is saved in to the notification repo of the passengers for later viewing. Again, when tracking the scenario, it amounts to 28 requests, and

again as they are fairly simple the hypothesis was the same, and was correct. The requests that took the longest across all prototypes were again fetching .js and .css data. This test case can be viewed as a proof that response time in a simple case like this, under consideration of one active user, is simply a testimony to the overall performance of the system, regardless of which underlying data management pattern it employs.

A reasonable remark could be to test with more active users; however, this was not done as it does not represent a realistic scenario within the context of the system. The flights were meant to be managed by an admin; whose rights are usually held by a single person. The performance test is nonetheless a testimony to the system itself.

5.3.3 Test Case IV: User login and flight booking

Prototype No.	No. of failed requests	RT under 800ms	RT under 1200ms	RT over 1200ms	Min. RT (ms)	Max. RT (ms)	Mean RT (ms)
1 active user (14 requests)							
I	0	13	0	1	17	1534	258
II	0	14	0	0	4	319	51
III	0	14	0	0	6	204	38
100 active users (1400 requests)							
I	0	1178	52	170	3	5404	557
II	0	1068	12	320	3	16807	1275
III	0	1212	24	164	3	5476	587
1000 active users (14000 requests)							
I	0	1752	125	12123	3	51099	8783
II	1972	66	30	10626	24	60026	21547
III	0	194	489	13317	512	56482	13101

Table 3: Gatling response time (RT) measurement overview

User login and flight booking test case involves the scenario of the first test case described and additionally includes the booking of a single flight. The booking of a flight requires filling in the booking, luggage and payment form, checking of the payment method in the payment microservice and showing the user the notification if everything went okay and booking was successful. This test case involves all of the microservices and is hence the most complex out of the four that the system has been tested with. When going through this scenario and tracking the information exchange, it amounts to the total of 14 requests. From the overview of the response times for each number of active users tested with (see Table 3), it can be seen that even with one user the third prototype performs better than it did for previous test cases, i.e. the question is to be posed if the benefits of event driven data management can be seen only on a more complex use case, when keeping scaling the same. When we scale

to a 100 or even better a 1000 of active users at once, we can see the third prototype taking a significant turn for better performance, and having served more requests under 800 milliseconds and having served requests without failure. All the requests that failed, failed due to a timeout of 60000 milliseconds (60 seconds) and failed when testing with the second prototype, indicating that the shared database scales poorly, as expected. Still in the lead is the first prototype, presumably due to lower overhead of decoupling just with a database-per-service approach, in comparison to using Kafka on top of that implementation. The scaling trend of the third prototype is however noticeable from this test case and it testifies to the assumption that when scaled out even more system could greatly benefit from the event-driven data management.

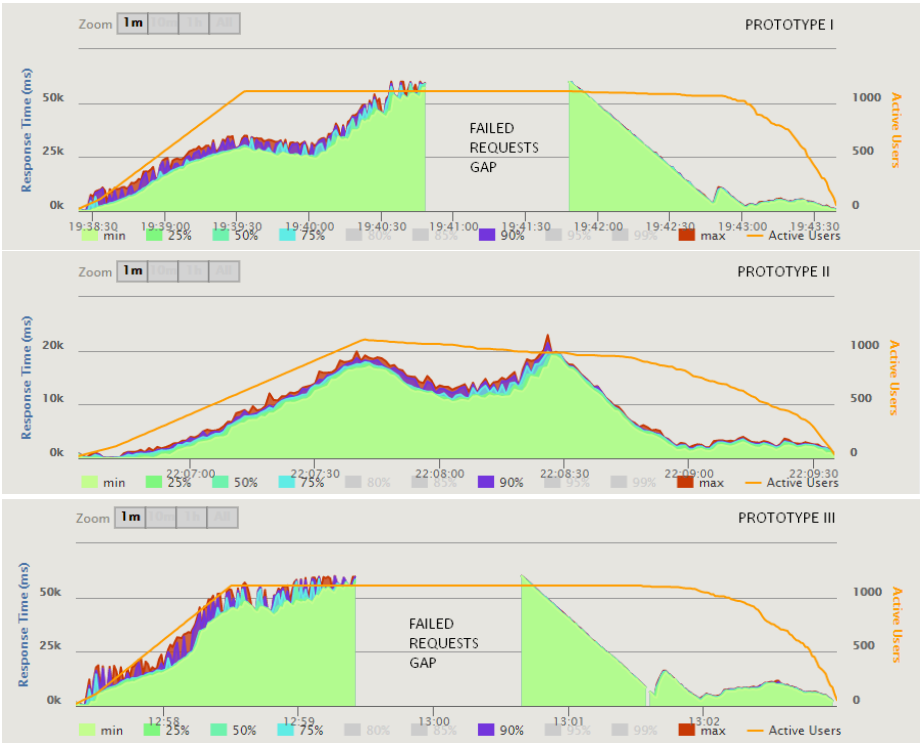

Figure 12: Response Time percentiles over time

If we take a look at the response time (see Figure 12) percentiles in the case where the simulation is started with 10 users, has 10 new each second over the next 10 seconds, and then ramps up to a 1000 over the following 60 seconds, it can be seen that the performance of first and third prototype is fairly similar. In both cases when the number of active users stays steady for a while, the system fails to serve all the requests simultaneously and due to a timeout of 60 seconds, the first prototype has 1136 failed requests and the third prototype 1357 failed

requests. Since this is the most load that the tests have been executed with and the test case is the most complex out of the four done, these results might be a consequence of combination of a high load of database-per-service and Kafka implementation with the hardware of the computer the tests were done on. It is plausible to think that if the same tests were repeated on a computer with more resources the results would be better. The second prototype, even though serving most requests over 1200 milliseconds, managed to serve all of them, but it is also clear from the graphs that the number of active users was less steady than in the other two cases, hence the number of requests was smaller at the given time. Considering the way, the prototypes acted in other test cases as well as in this one with a smaller load, it is clear that the start-time of the second prototype is better, but the constant scaling is where it starts to fail. Therefore, to a certain extent this behaviour is expected, as the quicker start once the simulation is started with 10 users only enables the quicker serving of requests during the period of time until the users start to scale more significantly. This quick start enables minimal hold of requests, and allows for quicker serving of the upcoming ones. The other two prototypes however, have a slower start and when they are about to catch up, the requests most on hold start timing out. But it is clear that the mean response time is smaller in a shared database implementation, which is surprising. This can however be an indication that some services exchange larger amounts of data and it might be beneficial to reconsider the decomposition of these services as well as partial implementation of shared database pattern over these services.

6 Conclusion

Within the scope of the thesis presented, a simple flight management system was developed to test most common approaches to data management in microservice-based applications. First prototype implemented the database-per-service pattern, where each of the microservices managed its own data privately and not allowing access unless it was through an API gateway. The prototype performed very well under a small load of users on the system and performed well under higher number of users but when tested on cases where communication with all of the services was not present. When scaling on a more complex test case the prototype had trouble serving all of the requests without timeout, meaning both response time and error rate were higher than desirable. Second prototype implemented the shared database pattern, where all of the services access the same database for the purpose of serving requests from the user i.e. gateway. The prototype performed very similarly to the first prototype under a small load, meaning the performance was satisfactory. However, on a larger load in a fairly simple test case of login and browse through the flights it had difficulty serving all of the requests due to a timeout, whereas the other two implementations had no such difficulties. On the most demanding test case however, only this prototype managed to serve all of the requests, even though the response time for most was over 1200 milliseconds. This is presumably due to a quick start that the shared database implementation showcased throughout testing phase. Third prototype implementing the event-driven data management performed the best under a steady high number of active users and its overhead highly likely influenced its performance on a smaller scale. The implementation of the prototypes and their testing is hardly a testimony to the properties of data management patterns in general, however it is a good testimony to the fact that there is no single correct data management approach when it comes to implementing a microservice-based application. Each system will benefit from a different data management solution and the solution should be tailored to the architecture of the system in question to achieve the most performance efficient and cost-effective implementation. In the case of the case study considered in this thesis, the system tested, seems to benefit the most from database-per-service pattern as it ensures data consistency, security and preserves the benefits of having a microservice based application. If the architecture of the system was organized differently so it requires less continuous input form the user, the system, especially when scaled could benefit greatly from the event-driven data management as well. However, to conclude with a significant degree of confidence which data management pattern or which combination of them is most beneficial for the system in question, future work is necessary.

6.1 Future Work

There is a number of further steps that can be taken to potentially improve the system in question and to test the effects of data management patterns on its performance and thereafter choose the best implementation approach. The ar-

chitecture of the system can be reconsidered and it can be tailored to request less constant input from the user which could go in line with a more cost-effective implementation of event-driven data management pattern as larger portion of request serving could be delegated to the internal microservice communication. Another step would be to reconsider the way in which the services were organized, whether there is a decomposition that delivers the functionality more efficiently under the three data management patterns in question. It is worthy of exploring if a hybrid i.e. a combination of patterns would suit the system more. To test the database-per-service pattern in more detail and see how performance effective it can actually be it is worthy to investigate if it could benefit from polyglot persistence of data i.e. testing different database types for different microservices to suit their needs and scaling more. The tests should also be done with a higher load and most importantly on a hardware with higher resources which would simulate the real-life scenarios more and detach the liability of the overhead of starting Kafka for example, as much as possible. In summary, there is many further cases that could and should be considered to say with determination which data management pattern is the best choice for the system in question. None of the scenarios however, would refute the fact that there is no single correct approach and that the performance of pattern chosen for any of the microservice-based application considered is highly dependent on the application itself i.e. the evaluation of a data management pattern is hardly ever detached from the application it is being implemented in.

References

[1] Data considerations for microservices. https://docs.microsoft.com/en-us/azure/architecture/microservices/design/data-considerations.

[2] Monolithic vs microservices architecture (msa). https://www.bmc.com/blogs/microservices-architecture/.

[3] Service discovery in a microservices architecture. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture.

[4] Chris Richardson of Eventuate, I. Building microservices: Inter-process communication in a microservices architecture. https://www.nginx.com/blog/building-microservices-inter-process-communication/.

[5] Chris Richardson of Eventuate, I. Building microservices: Using an api gateway. https://www.nginx.com/blog/building-microservices-using-an-api-gateway/.

[6] Chris Richardson of Eventuate, I. Event-driven data management for microservices. https://www.nginx.com/blog/event-driven-data-management-microservices/.

[7] Fowler, M. Microservices. https://martinfowler.com/articles/microservices.html.

[8] Kappagantula, S. Everything you need to know about microservices design patterns. https://www.edureka.co/blog/microservices-design-patterns#APIGateway.

[9] Kumar, R. Selecting the right database for your microservices. https://thenewstack.io/selecting-the-right-database-for-your-microservices/.

[10] LUMETTA, J. Monolith vs microservices: Which is the best option for you? https://www.webdesignerdepot.com/2018/05/monolith-vs-microservices-which-is-the-best-option-for-you/.

[11] McAllister, N. An introduction to event-driven architecture and apache kafka. https://tanzu.vmware.com/content/intersect/introduction-to-event-driven-architecture-and-apache-kafka.

[12] Peck, N. Microservice principles: Smart endpoints and dumb pipes. https://medium.com/@nathankpeck/microservice-principles-smart-endpoints-and-dumb-pipes-5691d410700f#.

[13] Richardson, C. *Microservices Patterns: With Examples in Java*. Manning Publications Co.

[14] Richardson, C. Pattern: Database per service. https://microservices.io/patterns/data/database-per-service.html.

[15] Richardson, C. Pattern: Event-driven architecture. https://microservices.io/patterns/data/event-driven-architecture.html.

[16] Richardson, C. Pattern: Microservice architecture. https://microservices.io/patterns/microservices.html.

[17] Sandra Tucker, B. A. Data management considerations for microservices solutions. https://www.ibm.com/garage/method/practices/code/data-management-for-microservices/.

[18] Schmitz, D. 10 tips for failing badly at microservices. https://www.youtube.com/watch?v=X0tjziAQfNQ&ab_channel=Devoxx.

[19] Singh, J. The what, why, and how of a microservices architecture. https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9.

[20] STOPFORD, B. Using apache kafka as a scalable, event-driven backbone for service architectures. https://www.confluent.io/blog/apache-kafka-for-service-architectures/.

[21] Tilkov, S. microxchg 2018 - microservice patterns antipatterns. https://www.youtube.com/watch?v=RsyOkifmamI&ab_channel=microXchg.

[22] Yehuda, Y. Microservices and databases: The main challenges. https://www3.dbmaestro.com/blog/microservices-and-databases-the-main-challenges.