Investigate availability and maintainability within a microservice architecture

Claus Djernæs Nielsen

Student number: 201300064

Master Thesis

14 June 2015

Supervisor: Henrik Bærbak Christensen,

Department of Computer Science, Aarhus University

AARHUS UNIVERSITET

Abstract

The microservice architectural style is a way of building software systems. A microservice system consists of many small services that communicate with each other over a network. Each service has an individual software development lifecycle. To work correctly, services depend on each other.

This thesis intends to evaluate different availability- and maintainability tactics through building a microservice prototype. The prototype is based on an existing use case, from the monolith application *"Customer Sales System" (KSS),* currently in production at Alm Brand.

A combination of a circuit breaker component and bulkheads are implemented to ensure a high level of availability. To reduce maintainability costs a microservice framework is implemented as a way to shield and share common functionality.

Various ways to handle distributed transactions in a microservice environment will be investigated and one of them selected as part of the implementation.

Contents

1. Motivation

Alm Brand is currently building all of their in-house applications based on Oracle's Fusion Middleware stack, relying on a Service Oriented Architecture (SOA) environment, and has been so for a number of years.

This has resulted in a large amount of applications and components that are closely related in a web of dependencies in a number of monolithic style systems. Making development, testing and deployments both risky and time consuming as the smallest changes may result in a need to build, deploy and test the entire monolith.

In order to handle these risks, which Alm. Brand has acknowledged are only increasing as more and more applications are introduced into the company. A simplification and an isolation of the various components are considered, to minimize the current level of dependencies between the components that together make up the backend of Alm Brand's system architecture.

By leaning towards a microservice architecture, the intention is to develop the components as self-contained units, which are developed, tested, deployed and run isolated from the rest. They hope this will minimize the time it takes to develop, maintain and test a component and the risks that might follow deploying it.

1.1. Characteristics of a Microservice System

No formal definition of microservices currently exists. The table below shows the common characteristics of a microservice, based on my findings from [1] and [8].

Componentization via services	Components in a microservice system expose their functionality through services. A service is a small self-contained application, which react to one or more actions as defined by its service contract.
Organize teams around business capabilities	A service is developed and maintained by one team and handles an isolated and well-defined part of the system.
Products not projects	A microservice is not handed over to a maintenance department when it is considered delivered, but owned by the same team throughout its entire lifetime.
Decentralized Data Management	Each microservice has its own data store, which cannot be accessed directly by any other component or microservice. All data access is achieved through functionality exposed by the owning microservice.
Multi-Language Capability	A microservice may be implemented in any programming language and use any storage technology the team may prefer. Meaning its not confined to the same technology choices as the rest of the system.
Continues deployments	Can be build, deployed and tested independently from each other.
Self-contained	A service is 100% self-contained, meaning it does not share resources or have any direct dependency to other parts of the system.
Out of service communication	The service runs in its own process and only communicates through messaging systems like RabbitMQ, typically using request/reply protocols.
Single Responsibility Principle	A microservice should perform a single business function, which implies that all of its components are highly cohesive. This means that a service should have one responsibility and only one.

Table 1.1 The characteristics of microservices [1] & [8].

1.2. Characteristics of a Monolithic System

Even though Alm Brand is striving for reusable components, they have repeatedly compromised a generic solution to a product specific one. This has resulted in a number of monolithic style systems, which in general has the characteristics listed below, partly based on [1] and [10].

Single codebase	The system is typically located within one large codebase. As the application grows so does its codebase. Quite possibly the team maintaining it as well.
Specific code language	Because of the single codebase characteristic, the entire system is usually built using one programming language.
Horizontal scaling	It is not possible to scale certain components within the monolithic application. It is the entire monolith or nothing. Scaling is accomplished using a load balancer and running several instances of the entire monolith application.
3-tier architecture	Client-side user interface (consisting of HTML pages and java script running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application.
Single process	A monolithic system like the server-side application mentioned above, typically executes in a single process.
Entangled components	Components in a monolithic system does not have any clear ownership, typically resulting in high coupling.
Long deployment cycles	Maintaining and extending the monolithic applications functionality is time consuming. Even the smallest change can result in a need to build, deploy and test the entire application.

Table 1.2 The characteristics of a Monolithic system [1] & [10].

1.3. Problem specification

This master thesis will investigate availability- and maintainability tactics within a microservice system.

A small microservice prototype is build based on the existing use case “Create meeting from Absalon” from the monolithic system Customer Sales System (KSS), identified in cooperation with Alm Brand.

The prototype will make up the foundation for investigating ways to ensure high availability within a highly decoupled system.

A microservice system might frequently expand with new services and over time consist of a large amount of smaller independent microservices, which may result in heavy maintenance.

The amount of time and workload a system requires in maintenance after its gone live has a large impact on its business case and expected benefit value. The maintenance cost of a project during its lifetime is typically estimated between 40-80% [28], often making maintenance exceed the initial development costs. Therefore, it is important always to consider the level of time a system will take to maintain and extend.

A system’s microservices are likely to contain a certain amount of similar functionality. Therefore, I intend to look into ways of wrapping that functionality so it can be shared across all services making maintenance of the microservice system simpler and less costly.

Specifically, I want to:

- Investigate availability tactics for the KSS use case using a microservice architecture. Specifically I will experiment with circuit breaker and bulkhead implementations and compare the resulting availability with that of the existing KSS architecture.
- Investigate modifiability tactics for the KSS use case, focusing on minimizing maintainability costs through designing and implementing a generic microservice framework.

1.4. Method

The main method used in this master thesis will be the prototyping iterative model.

Using prototyping will help me get experience with the microservice architecture style early in the process and at the same time highlight issues that needs clarification early, before they turn into showstoppers.

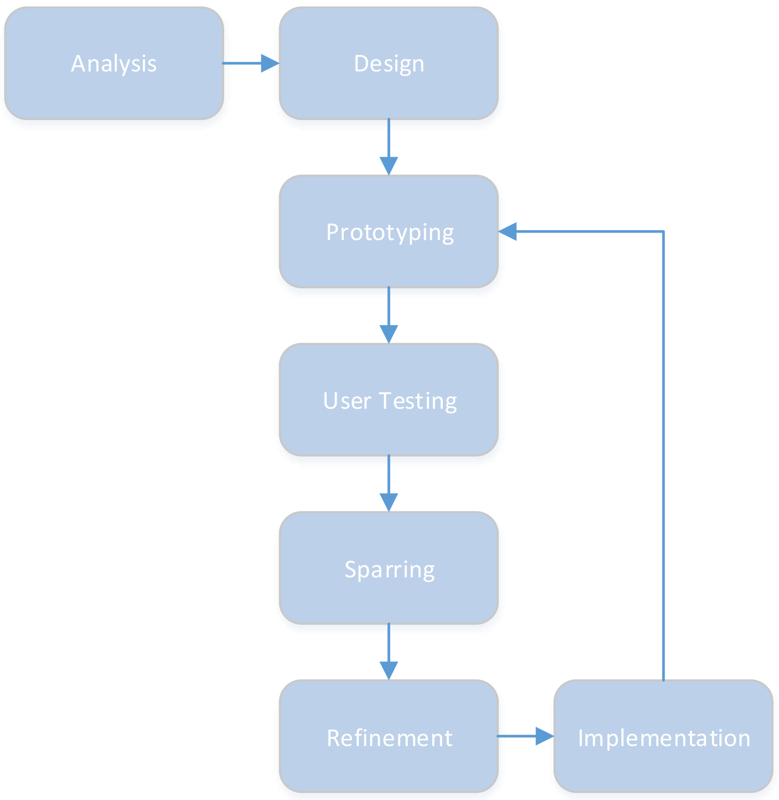


Figure 1.3 The prototyping iterative model. [2]

I have added the extra step Sparring, not originally suggested by the Illinois Institute of Technology [2]. As part of the sparring step, I intend to discuss and review my findings with Alm Brand, knowing they will challenge my conclusions and decisions. As I am a group of one, they will act as reviewers and sparring partners throughout this master thesis.

1.5. Definitions

Definitions used throughout this master thesis are mentioned below.

- **Customer Sales System/Kunde Salgs System (KSS)** is the telemarketing centers main tool, which has just gone live. It provides, received from Alm Brands BI solution, its users with lists of "cold leads" meaning potential customers and "hot leads" which is existing customers who might be interested in campaigns resulting in additional sales. The users call the leads and based on feedback, book meetings on behalf of and in the assurance agents calendars.

- **Monolith/Monolithic** is a single logical executable. Any changes done to one part of a monolithic system involves building and deploying a new version of the entire system. In regards to KSS it means that the different SOA components suffer from high coupling, making re-use difficult and even the smallest changes can affect several components.

- **Service oriented architecture (SOA)** is a technique that involves the interaction between loosely coupled services that function independently. A way of designing, developing, deploying, and managing systems, to achieve flexibility and scalability in large enterprise applications.

1.6. Delimitations

This master thesis is subject to the following delimitations.

- **Java-based prototype** - This prototype is built entirely in Java. For that reason, it will not show any potential issues with a polyglot system.

- **Company specific requirements** – Alm Brand has a number of in house technical requirements ex. specific security technology on all HTTP communication, all developer tools and technology choices are closely related to Oracle's Technology Stack etc. These requirements are considered. However, not necessarily taken into account.

2. Related work

Under related work, I will provide the reader with enough background information to understand the environment setup, patterns and technologies that forms the basis for decisions and considerations in later chapters.

2.1. Apache Kafka

Services within a microservice system typically communicates through message queues. This provides a loose coupling between the services and high flexibility in terms of system expansion, configuration and built-in surveillance options.

There are loads of messaging systems available; I have previously worked with RabbitMQ, Oracle JMS and Apache ActiveMQ. Oracle JMS is currently in use at Alm Brand, and therefore I know it can handle the load expected.

A messaging provider often mentioned together with microservices is Apache Kafka [11, 12, 13]. Apache Kafka is new to me, and this prototype gives me an opportunity to broaden my palette of tools. It can also serve as another technology suggestion for Alm Brand going forward, in case it turns out as a good solution.

Apache Kafka is generally considered to be a fast, scalable and distributed public-subscribe messaging system. By some the next generation messaging system, originally developed by LinkedIn and later adopted into the Apache Software Foundation [11].

Apache Kafka is selected because of its reputation as one of the best performance message queue providers out there. However, it should preferably be easy to replace with another provider, like Oracle JMS or RabbitMQ.

2.2. Distributed Transactions

Using transactions in "traditional" n-tier systems like a monolithic system is relatively simple. For example, when you run a transaction and an error or fault occurs, the transaction is aborted and any changes can easily be rolled back. Getting back a system-wide consistency.

When splitting a system into independent services like microservices, extra considerations comes into account.

What happens if some internal service transactions fail, while others succeed? – This could result in inconsistent data across the involved microservices. Another consideration has to do with getting some commitment from the services so the caller application can continue its work based on that commitment.

During system design, I have considered different approaches to solve the issues mentioned above.

2.2.1. Two-phase commit

One way, and a widely used approach, is the two-phase commit where the coordinator first checks if all participating services are ready to commit. Based on results/voting, ensuring uniformity to abort or commit at all services.

The greatest disadvantage of the two-phase commit protocol is that it is a blocking protocol. If the coordinator fails permanently, some services will never resolve their transactions: After a service has sent a *voting* message to the coordinator, it will block until it receives a *commit* or *rollback* request. [21].

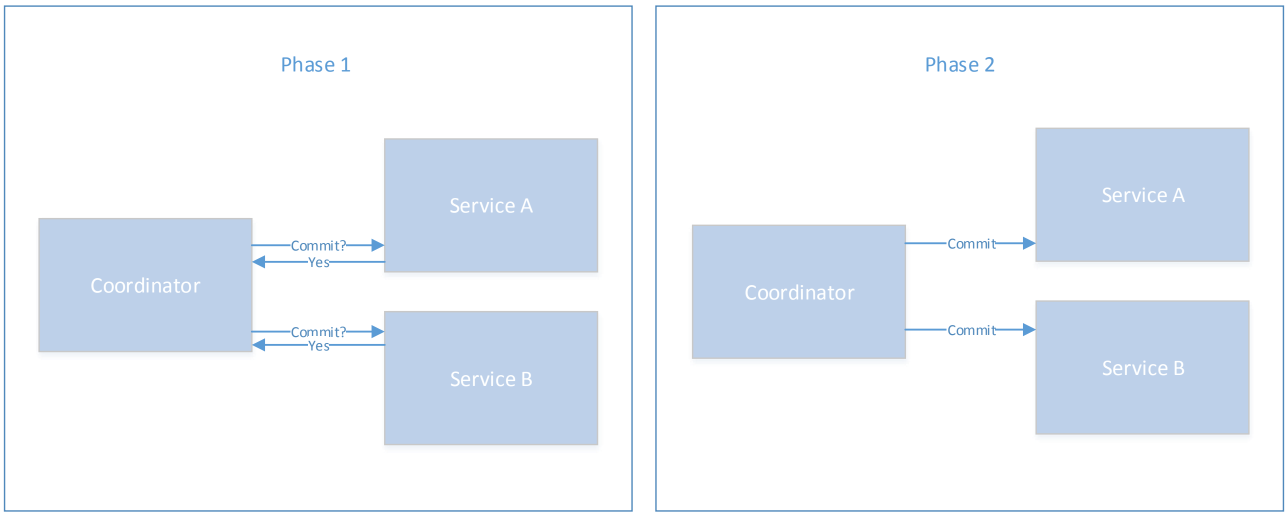


Figure 2.1 Two-phase commit

2.2.2. The Saga pattern

Another approach to achieve consensus between services is the Saga pattern.

This pattern consists of a coordinator service; in this setup, it would be located within the main caller component, which has the responsibility of controlling the involvement of the microservices. In order for the pattern to be applicable, the operations invoked on each participating service must all have a compensation operation. If one or more participating services fail while processing an operation, the coordinator will request the remaining participants to perform a corresponding compensation operation. [16] [17]

This means there is no rollback functionality if something goes wrong, instead all services must provide a compensation functionality, which reverses whatever action was just performed, as a kind of pseudo rollback.

This looks like a potential solution; however, there are a number of problems with compensations. These problems come from the fact that the changes made by the saga participants are not isolated. The lack of isolation means that other sagas may interact with the participating services operating on the same data that was just modified, making the compensation impossible.

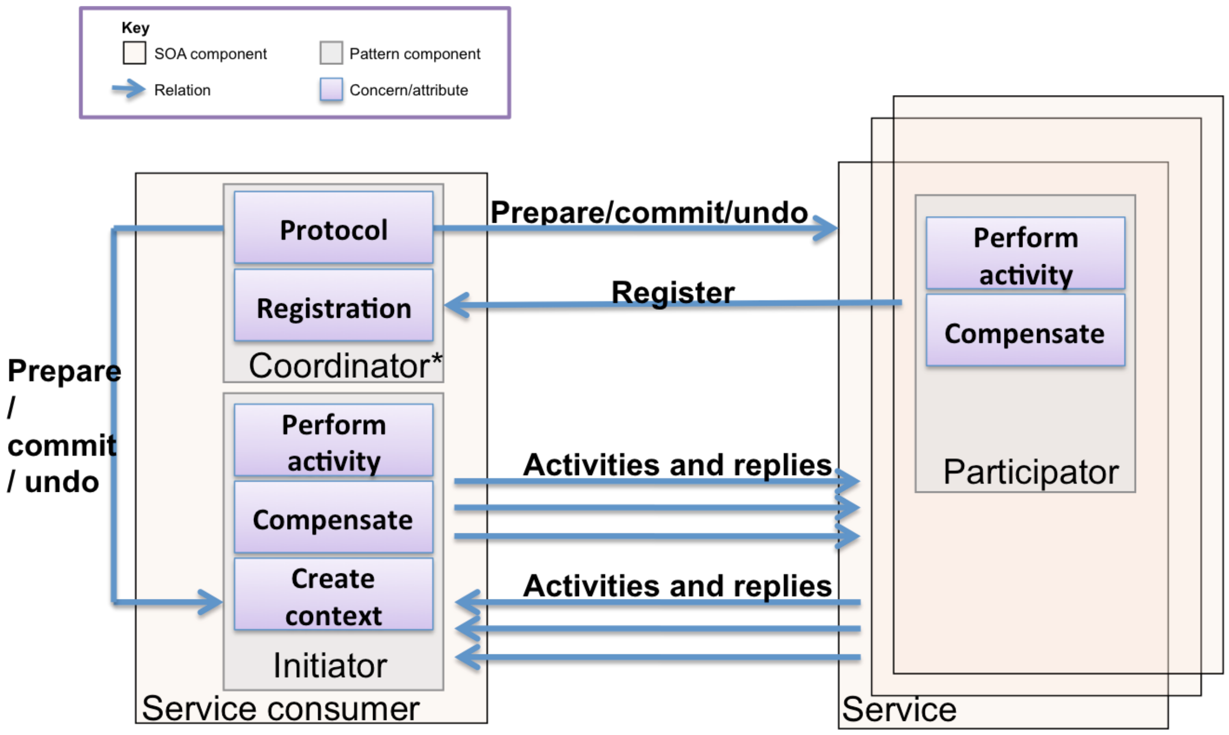


Figure 2.2 Saga pattern [17]

2.2.3. The Reservation pattern

The reservation pattern is a third approach for reaching a consensus between services, which unlike the Saga pattern does not include a central coordinator.

When a client needs to perform an action on one or more services, it does so by sending tentative events. These events are considered reservations, and are replied with a reservation id and an expiration date. The requester then has until the expiration date to confirm or cancel the operation.

A designated process residing within each service controls the reservations. This process handles all cancellations and confirmations as well as reservation expirations. In the case that a reservation expires, the process and its reserved resources are released.

In section 2.2.1, I mentioned the two-phase commit. The reservation pattern can be categorized as a two-pass protocol.

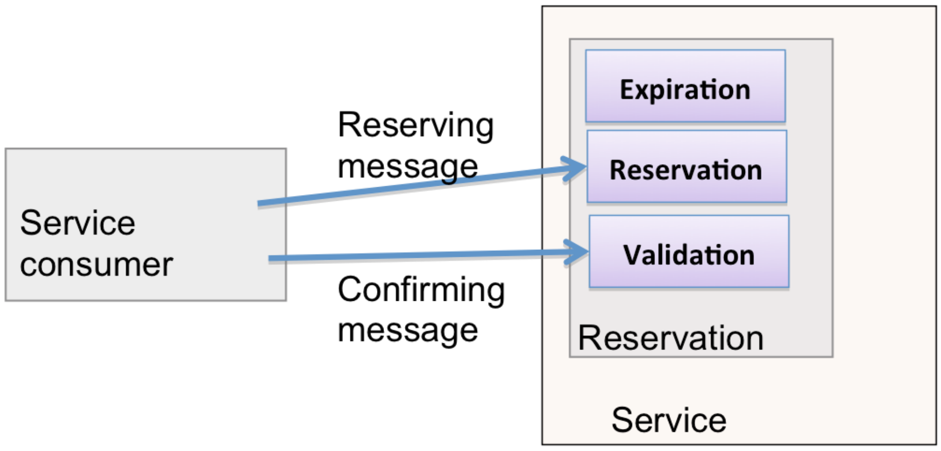


Figure 2.3 Reservation pattern [27]

The internal reservation component has the following responsibilities [22]:

- **Reservation** is making the reservation when a message that is considered "reserving" arrives. In addition to performing some actions like updating a database, it also sets a timer or an expiration time as part of the request confirmation.

- **Validation** makes sure that a reservation is still valid before finalizing the process.

- **Expiration** is marking invalid reservations when the conditions change. E.g. if a higher priority requester wants some data which is already reserved, the system can overrule the initial reservation and request it. It should also invalidate the initial reservation so when it is reclaimed, the system will know it is gone.

  Expiration can also be timed, as in "the data is reserved for 24 hours".

3. Customer Sales System (KSS)

The KSS Server component is part of the larger application Costumer Sales System (KSS), which has just gone live. Customer Sales System (KSS) is the telemarketing centers within Alm Brand's main tool used by the phoners.

A typical flow of the application starts with Alm Brands BI solution that delivers a daily list of "cold leads", which is potential customers to KSS. KSS also receives campaigns and "hot leads", meaning existing customer lists from a backend system called TIA.

The phoner get one lead displayed at a time who the phoner calls and tries to setup a meeting on behalf of an insurance agent. During the call, the phoner fills out a number of information about the lead. Based on the feedback the phoner books a meeting in the insurance agents' calendar. The meeting is booked in both KSS and IBM Notes, which is Alm Brands internal mail and calendar system. After this, the phoner clicks a button to get the next lead and the flow starts over.

Before the meeting, the insurance agent uses a different system called Absalon, to see what the phoner has written and agreed during the call with the lead.

3.1. Current component architecture

This section offers a short description of the components within the existing KSS system.

3.1.1. KSS Server

KSS Server is the main component and backend of KSS. It offers all of its functionality as a range web services. All requests and interactions with and in the KSS application goes through the KSS Server.

The KSS Server also integrates with a number of systems within Alm Brand. Like IBM Notes to book meetings in Alm Brand's in house calendar and mailing system. TIA to access existing customer data and register new customers. BAM that is an analytics and monitoring system. Finally, all events in KSS are stored using an Event component.

3.1.2. KSS Telemarketing (TMS)

TMS is the frontend of the KSS application. It integrates and gets most of its data from the web services exposed by the KSS Server, but in certain cases and for historic reasons access the same underlying KSS database directly. The typical workflow described above is done using this component.

3.1.3. Absalon

Absalon is a frontend application and the insurance agents' primary tool. It contains a dedicated KSS component, which integrates with the KSS Server, to display and modify the agents KSS calendar and information about leads.

3.1.4. KSS Mobile

A mobile version of the same KSS component functionality found in Absalon. KSS Mobile is developed in Oracle ADF (JSF), which is the primary technology used for all Alm Brands frontend applications.

3.1.5. TIA Insurance Application

TIA is a legacy system, which hold basic information about all customers, their insurance policies etc. within Alm Brand.

3.1.6. Campaign Controller

The Campaign Controller collects and stores all customer-oriented campaigns. These campaigns are send to KSS and then matched to a customer database located in TIA, to identify hot leads.

3.1.7. IBM Notes

Notes is the mail and calendar application from IBM used at Alm Brand. Just another tool like Microsoft Outlook.

3.1.8. Business Intelligence System (BI)

BI supplies KSS with "cold leads" lists on a daily basis or by request. An external vendor supplies the lists, in the end the same as looking up a list of random numbers in the phone book.

3.2. Existing use case “Create meeting from Absalon”

The use case “Create meeting from Absalon” is one of the most complex use cases in KSS, and affects most of the components in the system. It is also one of the first and main use cases when a user interacts with the system, which makes it the ideal candidate for this prototype.

The obvious responsibility of this use case is to create a meeting and book an insurance agent’ calendar in both Notes and KSS’s own calendar.

“Create meeting from Absalon” can be initiated from both Absalon and KSS mobile.

3.2.1. Current flow/architecture diagram (Monolithic)

A flow and architecture diagram for this use case in the existing monolithic system is found below in figure 3.1. The diagram focuses on the external flow of the use case.

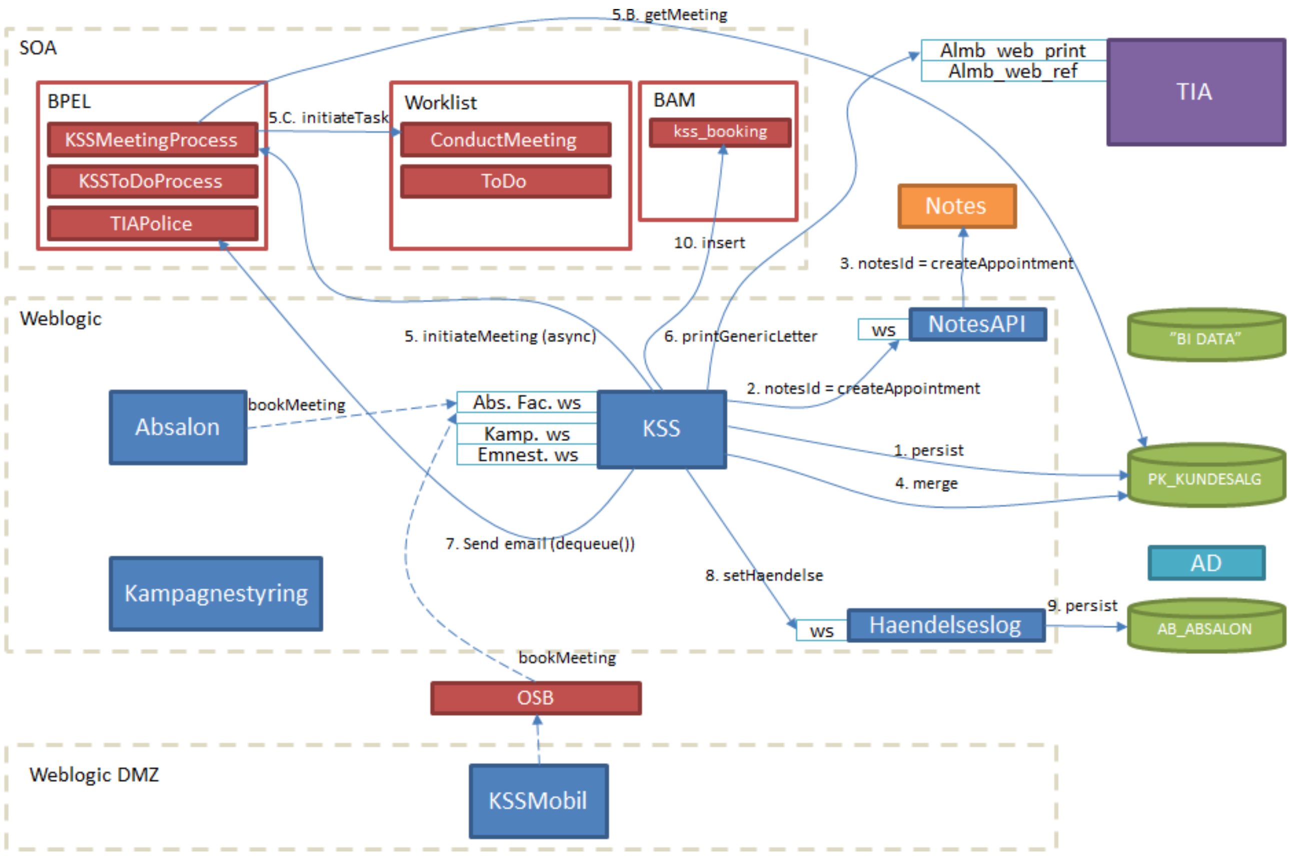


Figure 3.1 Use Case: Create meeting from Absalon (Monolithic flow diagram)

“Create meeting from Absalon” is initiated by calling the web service operation AbsalonFactoryWS.bookMeeting.

1. **Persist** creates an empty topic in the database, which is used throughout the use case. The topic consist of all data noted during the phoners’ conversation with the lead. The reason for creating an empty topic up front, is that the internal key (GUID) is used as an overall parent key for that specific booking.

Throughout the use case several events are logged in KSS's internal event log, which is not visible in figure 3.1.

Before step 2, an appointment is created in KSS calendar together with an empty meeting.

2. **CreateAppointment** creates a meeting in IBM Notes through the Notes API. (Step 1)
3. **CreateAppointment** same as above, just from Notes API to Notes. (Step 2)
4. **Merge** updates the topic with the latest data.
5. **InitiateMeeting** calls an external process, which asynchronously populates the empty meeting based on data collected from TIA and Absalon.
6. **PrintGenericLetter** is an external component that generates and sends a generic letter regarding the meeting to the lead, if they request it.
7. **SendEmail** sends a confirmation to the lead regarding the meeting.
8. **CreateHaendelse** logs the booking in Absalon's event log.
9. **Persist**, save event in Absalon's database.
10. **Insert** meeting in Oracle Business Activity Monitor (BAM) for real time event activity.

Most calls to third party resources like sendEmail and PrintGenericLetter are ignored in this use case, as they do not play a vital role in the architecture, and will initially be handled as part of the call sequence together with the rest of the microservices. BAM is included, to provide an example on how third party resources might be handled in a microservice style system.

4. Microservice system prototype

The prototype is the end software product of this master thesis. Through the prototype, I will investigate ways to improve and achieve high availability and low maintainability costs like mentioned in the problem specification.

4.1. Iteration 1

The existing use case “Create meeting from Absalon” consists of several microservice candidates, which I will come back to in chapter 4.2.2 Prototype component architecture. In this iteration, I have selected a couple of candidates in order to reach a certain level of complexity regarding the call sequence from the use case initiator. This will serve as a proper starting point and base for my investigations.

4.1.1. Goal

The main goal for this iteration is a partial version of the selected use case. It will provide hands on experience with the microservice architecture style, and will eventually be the first version of the prototype.

During prototype development, overall considerations on how to achieve the availability- and maintainability tactics mentioned in the problem specification will be described, and the solutions selected are implemented as part of the prototype.

4.1.2. Kafka integration

The Kafka server in this environment is installed on an Ubuntu OS running within a local VMware virtual machine.

A full example is included next on how to develop a producer and consumer using the Kafka API. The code in this section is a first version and will later be refactored and shared across all microservices through a common library for improved and simplified maintenance.

Kafka Producer

Below is the code for developing a simple producer that can publish text messages on a Kafka queue.

Producer configuration

In the Kafka producer, a simple configuration is used. It configures the broker, in this prototype only one Kafka broker/server is used. The configuration specifies timeout, serialization type and commit interval. Commit interval is how often the messages are saved to disk. Kafka does not send a message to a consumer before the message is stored on disk.

```
private ProducerConfig getKafkaProducerConfiguration() {
  Properties props = new Properties();
  props.put("metadata.broker.list", "ubuntu:9092");
  props.put("serializer.class", "kafka.serializer.StringEncoder");
  props.put("zookeeper.session.timeout.ms", "5000");
  props.put("request.timeout.ms", "5000");
  props.put("request.required.acks", "1");
  props.put("auto.commit.interval.ms", "1000");
  return new ProducerConfig(props);
}
```

Producer

The producer below uses the queue, in Kafka called a topic named “verification-topic”. It creates a KeyedMessage object, which can take three parameters. The topic, an optional key commonly referred to as a correlation Id and a message.

```
import java.util.Properties;

import kafka.javaapi.producer.Producer;

import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class KafkaProducer {
  final static String TOPIC = "verification-topic";

  public void publish(String topic, String key, String data) throws SystemException {
  try {
    KeyedMessage message = new KeyedMessage(topic, key, data);
    new Producer<String, String>(getKafkaConfiguration()).send(message);
  } catch (Exception e) {
    System.out.println("Could not send data: " + data +
                       " topic: " + topic +
                       " error msg: " + e.getMessage());
    throw new SystemException("Could not send data: "+data+" topic: "+topic, e);
  } finally {
    if(producer != null) {
      producer.close();
    }
  }
}
```

Kafka Consumer

The consumer starts as a separate thread running an infinite loop listening for new messages.

Consumer configuration

Below is the configuration for the consumer. The consumer connects to a zookeeper server, which distributes the consumers to the Kafka servers active in the cluster or group. The autocommit interval is the frequency that the consumed offsets are committed to zookeeper. This means that next time a message sequence is send to the consumer, it starts from the new offset value.

```
private ConsumerConfig getKafkaConsumerConfiguration() {
  Properties props = new Properties();
  props.put("zookeeper.connect", "ubuntu:2181");
  props.put("zookeeper.connection.timeout.ms", "1000000");
  props.put("group.id", "test-group");
  props.put("autocommit.interval.ms", "1000");
  return new ConsumerConfig(props);
}
```

Consumer

The consumer below shows the full code of the first version of the prototype Kafka consumer. It simply outputs any message fetched from the same queue “verification-topic” that the producer uses to publish messages.

```java
import java.io.UnsupportedEncodingException;

import java.nio.ByteBuffer;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;

import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;

import kafka.javaapi.consumer.ConsumerConnector;
import kafka.javaapi.message.ByteBufferMessageSet;

import kafka.message.MessageAndOffset;

public class KafkaConsumer extends Thread {
  final static String clientId = "ConsumerClient";
  final static String TOPIC = "verification-topic";

  public static void main(String[] arg) throws UnsupportedEncodingException {
    KafkaConsumer kafkaConsumer = new KafkaConsumer();
    kafkaConsumer.start();
  }

  @Override
  public void run() {
    ConsumerConnector consumerConnector =
      Consumer.createJavaConsumerConnector(getKafkaConsumerConfiguration());

    Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
    topicCountMap.put(TOPIC, 1);

    Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
      consumerConnector.createMessageStreams(topicCountMap);
    KafkaStream<byte[], byte[]> stream = consumerMap.get(TOPIC).get(0);
    ConsumerIterator<byte[], byte[]> it = stream.iterator();

    while (it.hasNext()) {
      System.out.println("message received: " + new String(it.next().message()));
    }
  }
}
```

4.1.3. Microservice implementation

During design and development, candidate operations, helper classes and components are considered for the microservice framework. The identified microservices are expected to evolve throughout this thesis alongside the framework.

Below is parts of the initial version of the Topic microservice, which is also used as the candidate to illustrate the progress in the next iteration.

I have chosen not to include any code snippets on how to integrate with the databases used in this prototype or the common business and validation code. This is because I do not consider this vital for the overall goal of this thesis, and therefore left out of the report.

Below is the initial implementation on how to initialize the framework. A configuration for both a Kafka producer and consumer is needed to both listen for requests and publish response messages to Kafka.

```
public class TopicService extends MicroserviceWrapper {
  private TopicServiceFacade facade = null;

  public TopicService(String producerHost, String consumerHost, String serializer,
                      String topic, String group, String surveillanceTopic)
                      throws ArgumentOutOfRangeException, SystemException
    super(producerHost, consumerHost, serializer, topic, group, surveillanceTopic);
  }
}
```

Both SOAP (XML) and JSON based messages are valid options for communicating between services. SOAP (XML) is easier to validate but contains more information overhead, which is the main reason for selecting JSON.

This snippet illustrates how easy Google GSON can be used to parse a JsonObject to an internal Topic object.

```
public class RequestObjectMapper {
  public static Topic toTopic(JsonObject object) {
    return new Gson().fromJson(object, Topic.class);
  }
}
```

The functionality in this first version does not include proper error handling, and is solely tested in a happy path manor. A later version of the onMessage operation will be added as part of the second iteration, which includes some degree of error handling and more.

```
public final void onMessage(Event event) {
  if(request.getOperation() == TopicServiceOperations.CREATE_OR_UPDATE_TOPIC) {
      return getFacade().createOrUpdateTopic(request);
  } else if(request.getOperation() == TopicServiceOperations.GET_ALL_TOPICS) {
      return getFacade().getAllTopics(request);
  } else {
      return EventMessageBuilder.buildResponseEvent(-1, request.getApplicationId(),
               request.getKey(), "Invalid Operation", Constants.APPLICATION_ID);
  }
}
```

The event object is parsed to JSON and send to the requested topic/queue.

```
public final void publish(Event event, String topic) throws SystemException {
  String message = "";
  try {

    if (topic == null || topic == "") {
      logger.error("Topic was empty...");
      throw new SystemException("No topic has been defined");
    }

    if (event instanceof ResponseEvent) {
      message = new GsonUtils<ResponseEvent>().toJSON((ResponseEvent)event);
    } else if (event instanceof RequestEvent) {
      message = new GsonUtils<RequestEvent>().toJSON((RequestEvent)event);
    }

    if (event != null) {
      if (message != null && !message.equals("")) {
        getKafka().getProducer().publish(topic, event.getKey(), message);
      } else {
        throw new SystemException("data object was null or empty");
      }
    }
  } catch (Exception e) {
    logger.error("Could not publish messageError: " + message, e);
  }
}
```

4.1.4. Circuit Breaker and Timeout

Michael T. Nygard [23, chapter 5] suggests several ways to improve the overall stability of a system. A combination of timeouts and the circuit breaker pattern also suggested by Martin Fowler [24] provides a high amount of protection from external calls increasing the stability and thereby the availability of the entire system.

The overall goal is to end up with an independent generic circuit breaker component, making it useful and applicable in several scenarios.

The idea behind the circuit breaker pattern is quite simple. The circuit breaker wraps operation calls into a component, which shields the caller from calling the real operation. It monitors for failures and if a specified failure threshold is reached, it blocks and returns an error. It does so for a certain amount of time, and then opens up for one call that determines if the failure count should reset or initialize a new failure period.

The pattern consist of three states: Closed, Open and Half Open.

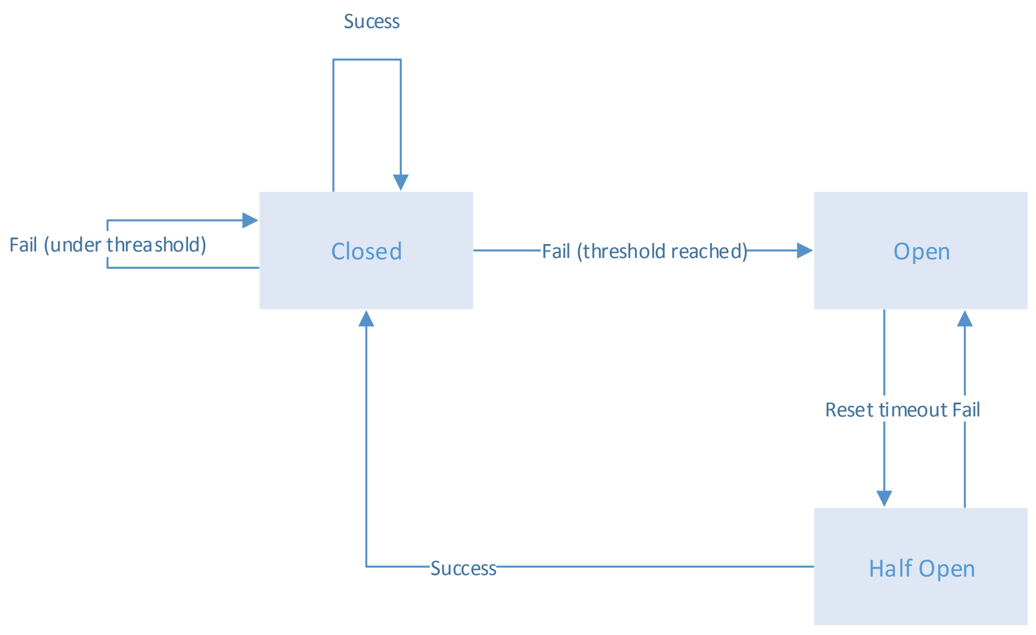


Figure 4.1 Circuit Breaker Pattern [24]

Closed

Closed is the initial state of the pattern. In this state all calls to the wrapped resource are accepted. If a call fails a failure counter is incremented, until it reaches a defined threshold. If the threshold is reached, meaning that the resource has failed consistently a predefined amount of times, the state changes to Open.

Open

When it enters the Open state, a timeout period is initiated. While In the Open state all calls are rejected and results in a raised circuit open exception. When the timeout period expires, the state is changed to Half Open.

Half Open

In the HalfOpen state, the next call to the wrapped resource is accepted and determines the following state. If the call fails, it moves back to the Open state and the timeout period starts again. If the call to the wrapped resource succeeds while in the Half Open state, it moves into the Closed state and resets its failure counter.

The circuit breaker component is located within the microservice commons library. Both the services and its clients use the library to access the Kafka queues, making it available to all actors in the system. At the same time it is also used within the library to shield the caller from the Kafka API calls like the publish method, used every time a message is put on a queue.

As part of the circuit breaker component, timeout handling is also implemented to ensure the resource only waits on a response for a configurable amount of time.

A timeout simply means that a process will not wait forever on some requested resource. The purpose of applying timeouts is to give quicker feedback to the users of the system as well as avoiding situations where parts of the system have exhausted resources due to waiting on others.

The full circuit breaker and timeout component implementation is available in appendix 9.3.

In short:

- The circuit breaker provides improved stability/availability by shielding the caller from calling the real operation when reaching a specified failure count.
- Timeouts ensures that the system does not wait forever on a response from an external resource, thereby also improving the overall system stability/availability.

4.1.5. Bulkheads

Because a microservice system by nature consist of several self-contained services that typically rely on messaging systems for communication, it automatically adopts the bulkhead stability pattern [23, chapter 5.3].

In general, the bulkhead stability pattern is a pattern, which partitions a system so failures in one part of a system does not lead to total system failures.

This means as the system evolves and more and more services are introduced several of them will likely be involved in various use cases. If one of them fails, it will only effect the parts of the system where that service is involved, and not cause the entire system to fail.

Using Kafka as message broker allows for multiple instances of each service. Each service may be replicated and deployed to different physical machines for improved system availability/stability.

Replicating and deploying services to different servers also support ways to solve what Nygard calls *Hidden Linkages* [23, chapter 5.3]. Hidden Linkages means that two separate systems use the same enterprise service creating a “hidden” vulnerability towards each other. If one causes the enterprise service to fail or perform poorly e.g. because of user load, it will at some point effect the other system.

A way to solve this is to replicate the services and have a dedicated service pool for each system, separating the client systems entirely from each other.

In short:

- Each microservice is running in separate processes, which ensures improved stability for all services since failures in one service cannot cause others to fail.
- Using Kafka as message broker allows for multiple instances of each service. A service might be replicated and deployed to different physical machines for improved availability/stability.
- Starting and stopping a microservice is simple and easy. This will help to scale the overall performance when the number of users increase.
- Bulkheads also adds support to prevent Hidden Linkages by replicating services, and make each system use its own dedicated service or service pool.

4.1.6. Experiments

Throughout this first iteration, a large number of experiments was conducted to verify that my implementation, design and chosen patterns support the end goal for the first microservice Topic and its primary operation create_or_update_topic.

A complete test suite has not been preferred to verify all possible scenarios, test boundary values etc. of the Topic service, the circuit breaker component and so on. Instead, my focus has been on testing the overall architecture and basic functionality of each component, and how they can support availability and maintainability as mentioned in the problem specification.

In appendix 9.4. It is possible to see an example of the full message flow of the operation create_or_update_topic from one of these experiments.

Entire stack traces for experiments performed as part of both the first and the second iteration can be found in the zip file here: http://users-cs.au.dk/baerbak/c/vm/cn_ms_prototype.zip, which also contains the source code for the entire prototype.

4.1.7. Partial conclusion

The focus was to develop a partial version of the existing use case “Create meeting from Absalon”. Both to gain experience with the microservice architecture style, and research solutions to ensure a high level of availability as well as ways to reduce the level of maintenance required across the microservices through functionality reuse.

Various patterns and solutions was considered to ensure high availability. I decided to implement a version of the circuit breaker pattern combined with timeouts suggested by both Nygard [23, chapter 5] and Fowler [24]. The circuit breaker component shields the application from external calls; this ensures a high level of protection from third party applications like Kafka. Resulting in a more robust system and thereby improving the overall system stability/availability.

Bulkheads are also introduced as a way to ensure high availability. Because of the architectural nature of a microservice system and its use of message queues for communication, the system automatically adopts the bulkhead pattern. The services are loosely coupled by nature, ensuring that an exception in one service does not affect the rest of the microservices within the system. Using Kafka as message broker also supports use of multiple instances of a service, which helps improve the systems overall availability/stability.

The development of a microservice commons library was initiated and is further extended in the next iteration. This ensures that all shared functionality is isolated within one library, both minimizing the amount of testing needed, but also helps to keep the development maintenance costs down for any library modifications. When the commons library is changed, the changes are automatically materialized down through the services that implements the library, which is expected to be all microservices within the system. A new deployment of each service is required though, in order for the changes to take effect.

Each microservice executes its own isolated transaction, which quickly revealed a crucial issue. What happens if one microservice transaction fails while the rest succeeds? – That will potentially cause the system to end in an inconsistency state.

4.2. Iteration 2

This iteration is an extension of the previous. The selected use case is expanded to include the rest of the identified microservices, an overview of these are added later in this section. At the same time, the declared goals in the problem specification regarding availability- and maintainability tactics are further clarified.

4.2.1. Goal

During this iteration, the prototype is expanded to include the rest of the microservices identified from the existing use case “Create meeting from Absalon”. I will also elaborate and provide solutions to issues that emerged during the first iteration.

The main issue I encountered was handling of distributed transactions between the involved services. Therefore, the primary goal for this iteration is to consider and implement a way to handle distributed transactions within a microservice system. In short, if one service fails, it is important that nothing is committed elsewhere. Otherwise, the system will end in a state of inconsistency.

Different candidate patterns to handle distributed transactions was described in section 2.2. Distributed Transactions and at least one of these will be implemented as part of the prototype.

Furthermore, I will extend the microservice framework that was initiated in the first iteration and provide the full architecture and implementation.

4.2.2. Prototype component architecture

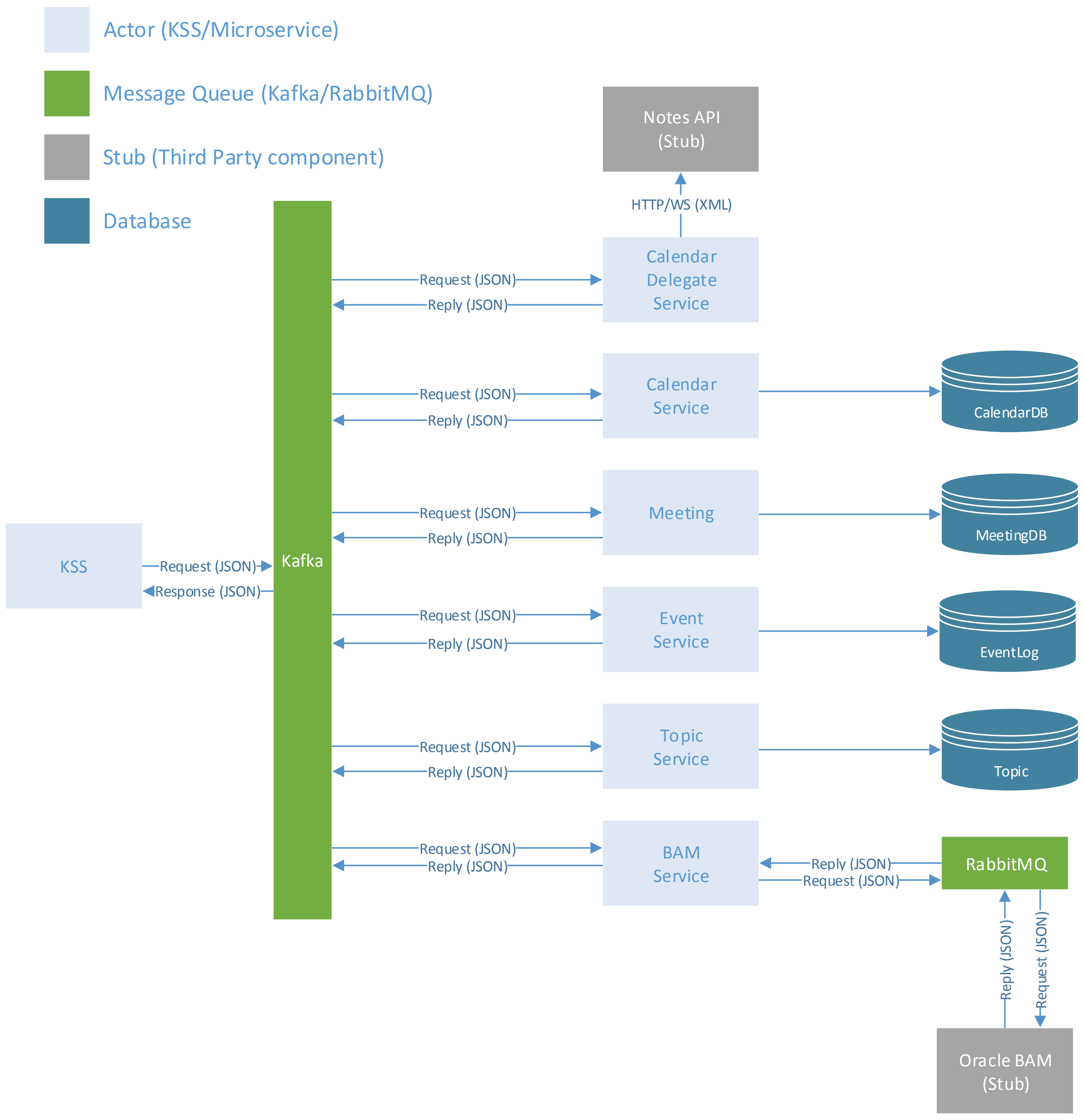


Figure 4.2 “Create meeting from Absalon” microservice architecture.

Based on meetings with KSS project stakeholders and code reviews of the existing KSS application I have identified the microservices shown in figure 4.2.

I originally had the meeting functionality as a separate component like shown above, as the meeting is created independently and used both by KSS but also directly accessed by Absalon. I therefore assumed based on the code reviews that it was possible to create a meeting, without booking a time slot in the calendar, kind of a temporary agreement.

After discussing the original design with Alm Brand, the meeting functionality is moved into the Calendar component. Because it does not currently make sense or considered possible to create a meeting without booking a timeslot in the calendar. This change will be visible in figure 4.4 KSS Server microservice architecture.

When designing microservices their responsibility and scope is determined based on their corresponding business capabilities.

This means unless I get a requirement from the business, requesting the possibility of creating a meeting without booking a timeslot, it would go against the mindset, to split calendar and meeting up. Should such a requirement come later, it might result in a service of its own, having meetings registered both in a separate service and in the existing calendar service, having the microservice style requirement of self-contained services in mind.

A meeting is always booked in Notes and in the Calendar service, never one or the other. This suggests encapsulating this functionality in a delegate calendar component, to simplify the integration and ensure a meeting is always created at both places.

Oracle Business Activity Monitor (BAM) is a real time statistics and reporting component used to display dashboards on big screens placed around the offices. Typically used by the management to see how many meetings has been booked the past 24 hours, in total or by each employee. BAM is a standard Oracle product that takes loads of input parameters to cover all possible requirements and scenarios. Each project in Alm Brand uses BAM, and every project has its own interface defined. By isolating the integration to BAM in a separate component, I achieve low coupling to the rest of KSS and the opportunity to simplify the interface, aimed at the way Alm Brand uses BAM, in this case the KSS project.

Topic is a cross functional component used by several actors in the system. KSS, Absalon and KSSMobile. Topic is an entity component that holds all agreements and information noted by the phoner during the call to a potential customer. A Topic can exist without a meeting, which is together with the number of actors using it, the main reason for an independent component.

There exist several event log components in KSS. The KSSEventlog is the internal log used to log all calls from KSS. This also handles blocking of leads. In case a lead does not pick up after four rings or the number is inactive, the lead is flagged in the event log.

The KSSEventlog functionality is already self-contained in a way. No direct relation to the rest of KSS exists, even though it currently shares the same code base and stores its data in the same database schema.

4.2.3. Distributed Transactions

During system design, I considered different approaches to solve this issue, which is further elaborated on in this section.

Because of the time constraint of this master thesis, the Reservation pattern has been preferred for implementation appose to a version of the Saga pattern. It might result in a more chatty protocol because of its two-pass setup. However, because of the simplicity of reservations and real life relation, it will be quicker to implement and simpler to present to non-technical staff, instead of the compensation functionality required by the Saga pattern.

Reservation pattern

Like mentioned in section 2.2. Distributed Transactions, dealing with many smaller transactions spread across several independent services is very different and far more difficult than having only one transaction running within the same scope.

Under 2.2. Distributed Transactions, I presented three ways to handle this problem. During system design, I have decided to focus on the reservation pattern.

Reservations happen in business transactions every day, making the concept and idea of the pattern easy to understand and describe to colleagues and stakeholders.

The reservation pattern consists of the three events, Reservation, Validation and Confirmation, described in section 2.2.3.

I considered several technical solutions for this pattern. I originally passed the raw events to the microservices making each of them handle the reservations and cleanup. This solution would allow each service to implement the pattern best suited to its responsibility and setup. However, it resulted in a lot of code duplication, having the same logic and responsibility split across all services in the system. I soon figured it to be a poor and time-consuming solution, for testing, maintenance and building new services, making me redesign the pattern setup.

Instead, I moved everything up into the microservice framework and stored the reservations in memory. This way none of the microservices has any direct access or knowledge of the reservations or pattern for that matter. This solution has a vulnerability that if the service goes down or is re-started, all reservations will be lost. Reservations would however only exist for a short period, maybe milliseconds, keeping the potential loss of reservations to a minimum. If the reservations was stored in a database allowing them to survive a system break down, the processes who created the reservations would probably have rolled back or otherwise terminated, before the service got back up, making this a viable solution for the prototype. However a more durable solution is needed going forward. First choice was to implement an in-memory database solution using Derby, this setup turned out to by unnecessary complex for the systems needs which lead to a simple cached hash map solution that can be found in appendix 9.1.

Implementing the reservation pattern introduces a number of risks that should be considered and handled. [22]

First of all a reservation is basically a lock, thus introducing the risk of deadlocks in a distributed environment.

For instance if two participants initiates the flow in reverse order, meaning that caller A reserves resource C and caller B reserves resource D, making A wait for resource D and B wait for resource C. This situation is resolved either by the callers themselves through timeouts, making them try again based on retries or the reservation cleanup logic. Nevertheless, if the callers continue to allocate the same resources repeatedly, it might actually result in a deadlock.

Another risk is the potential of Denial of Service. For instance if a service keeps re-reserving a resource, this could potential result in a DOA, and stall other services waiting for that specific resource. This risk will not be handled by the prototype.

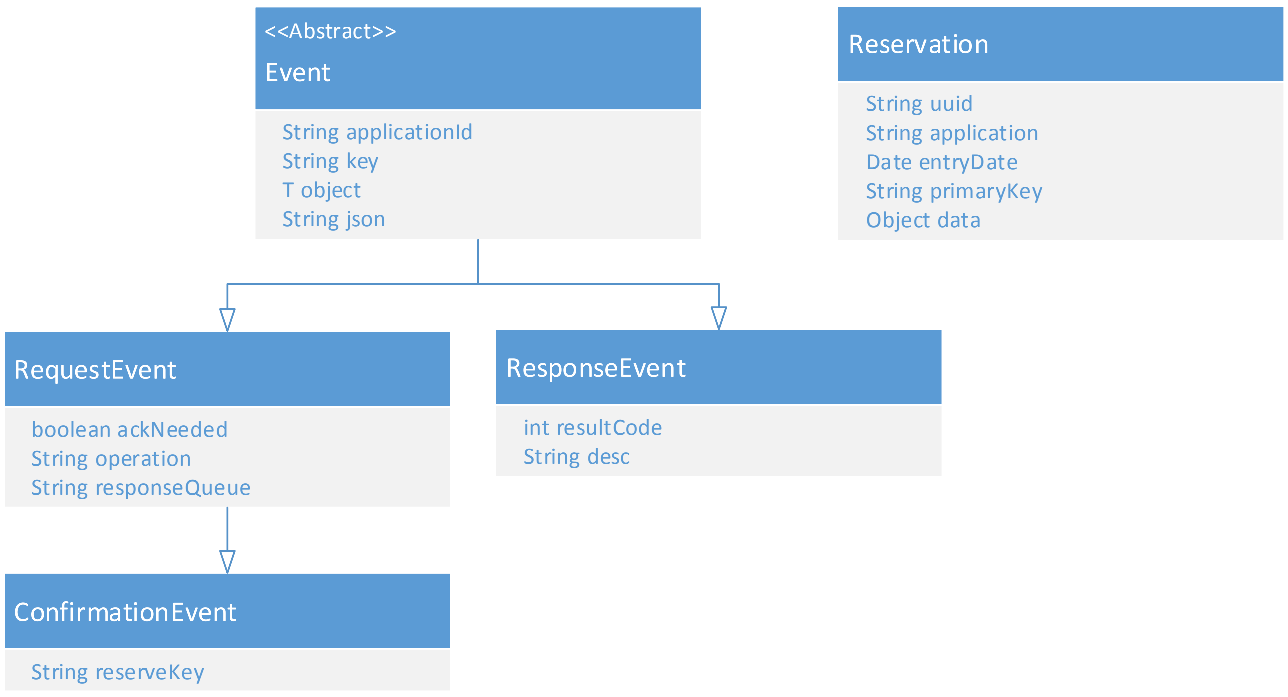


Figure 4.3 Reservation model

The overall entity class in the reservation model is the abstract class Event. Event contains a correlation key, applicationId, the raw JSON message string and an object, which is a google.gson.JsonObject of the message.

Besides the abstract class Event, pojo classes are added to represent and hold each specific event type.

If an incoming message cannot be parsed to one of the event types RequestEvent, ConfirmationEvent or ResponseEvent, the message is flagged as invalid, and added to a specific error queue for manual handling.

All reservations are stored as a reservation object in a Map. The correlation id is the key and the reservation holds the object's primary key, reservation timestamp and the updated data object

Implementation

The method handling all incoming messages and reservations can be found in appendix 9.2.

Each microservice must extend the abstract class MicroserviceWrapper located within the microservice base library. This provides access to the basic microservice functionality and requires each service to implement a number of methods used by the wrapper class for reservation handling, validation, cleaning and events.

```java
public class TopicService extends MicroserviceWrapper {

  private TopicServiceFacade facade = null;

  private final Logger logger = Logger.getLogger(TopicService.class);

  public TopicService(String producerHost, String consumerHost, String serializer,
                      String topic, String group, String surveillanceTopic) throws
                                       ArgumentOutOfRangeException, SystemException {
    super(producerHost, consumerHost, serializer, topic, group, surveillanceTopic);
    logger.info("Woohoo saying hallo to Topic Microservice!!!");
  }
…
```

The methods below as defined in the abstract MicroserviceWrapper class that each microservice must implement.

```java
/**
* Do nothing!! - Methods should be overridden by the final microservice
* implementation, to catch and handle incoming messages.
*/
public abstract boolean isUpdateEvent(RequestEvent event);
public abstract String getPrimaryKey(Event event);
public abstract int handleRequestEvent(RequestEvent event);
public abstract void handleResponseEvent(ResponseEvent event);
public abstract void handleConfirmationEvent(ConfirmationEvent event);
public abstract void onReservationCleanUp(List<Reservation> expired);
```

Below is part of the implementation for HandleRequestEvent from the Topic service. The first operation is the implementation of the handleRequestEvent operation, mentioned above. It is called by the microservice commons library when a request message is received.

```java
@Override
public int handleRequestEvent(RequestEvent requestEvent) {
  String responseTopic = "";
  ResponseEvent response = null;

  try {
    response = getFacade().handleRequest(requestEvent);
    responseTopic = requestEvent.getResponseQueue();

    if(response != null && responseTopic != null && !responseTopic.equals(""))  {
      publish(response, responseTopic);
    }

  } catch (Exception e) {
    logger.error("Error while handling message. Error: " + e.getMessage());
    logInvalidMessage(requestEvent, "Error while handling message.", "",
                                                        Constants.APPLICATION_ID);
  }
  return response.getResultCode();
}
```

The method below shows the service business layer called by the previously described method. Its sole responsibility is to extract the operation from the received message and delegate it to the proper operation functionality.

```
public ResponseEvent handleRequest(RequestEvent request) {
  TopicServiceOperations operation = TopicServiceOperations.EMPTY;

  try {
    operation = TopicServiceOperations.valueOf(request.getOperation());
  } catch(IllegalArgumentException e) {
    return EventMessageBuilder.buildResponseEvent(-1, request.getApplicationId(),
             request.getKey(), "Invalid Operation", Constants.APPLICATION_ID);
  }

  if(operation == TopicServiceOperations.CREATE_OR_UPDATE_TOPIC) {
    return getFacade().createOrUpdateTopic(request);
  } else if(operation == TopicServiceOperations.GET_ALL_TOPICS) {
    return getFacade().getAllTopics(request);
  } else {
    return EventMessageBuilder.buildResponseEvent(-1, request.getApplicationId(),
             request.getKey(), "Invalid Operation", Constants.APPLICATION_ID);
  }
}
```

This example shows the createOrUpdateTopic operation. Using the reservation pattern, nothing is created or committed at this step. The request input is validated, and based on the result a reservation is created and stored in memory by the microservice library. Here it waits for a confirmation event, which will create or update the topic.

All helper operations regarding messaging is located within the microservice commons library, like the EventMessageBuilder and ResultCodes, described in section 4.2.4. Microservice framework & architecture.

```
public ResponseEvent createOrUpdateTopic(RequestEvent request) {
  int resultCode =  ResultCodes.STATUS_RESERVATION_OK;
  String description = "OK";

  try {
    Topic topic = RequestObjectMapper.toTopic((JsonObject)request.getObject());
    TopicValidator.validate(topic);
  } catch (BusinessException e) {
    resultCode =  ResultCodes.STATUS_ERROR;
    description = e.getMessage();
  } catch (Exception e) {
    resultCode =  ResultCodes.STATUS_INVALID;
    description = "Invalid Type";
  }
  return EventMessageBuilder.buildResponseEvent(resultCode, description,
           request.getKey(), null, Constants.APPLICATION_ID);
}
```

Below is a snippet of the KSS server component that plays the role as the use case client.

This first snippet calls the operation CREATE_OR_UPDATE_TOPIC on the topic service:

```
KSSServerBusiness business = new KSSServerBusiness();
/** Topic **/
RequestEvent topicRequest = new RequestEvent();
topicRequest.setResponseQueue(KSSServerBusiness.TOPIC_RESPONSE_QUEUE);
topicRequest.setOperation("CREATE_OR_UPDATE_TOPIC");
topicRequest.setApplicationId("KSS_Server");
topicRequest.setObject(buildTopic());

business.getTopic().getProducer().publish(KSSServerBusiness.TOPIC_REQUEST_QUEUE,
UUID.randomUUID().toString(), new GsonUtils<RequestEvent>().toJSON(topicRequest));
business.incrementServiceCounter();
```

The method below builds an object of the TopicService client that extends KafkaConsumerListener located in the microservice commons library, shown later:

```
public KafkaFacade getTopic() throws ArgumentOutOfRangeException, SystemException {
  if (topic == null) {
    topicListener = new TopicListener(this);
    topic = new KafkaFacadeImpl(topicListener, "ubuntu:9092", "ubuntu:2181",
               "kafka.serializer.StringEncoder", TOPIC_RESPONSE_QUEUE, "test-group");
  }
  return topic;
}
```

The custom onMessage method in the TopicListener service client:

```
@Override
public void onMessage(String string) {
  try {
    ResponseEvent response = new GsonUtils<ResponseEvent>().fromJSON(string,
                                                                ResponseEvent.class);

    if(server != null) {
      server.reportServiceError(response.getResultCode());

      if (response.getResultCode() == ResultCodes.STATUS_RESERVATION_OK) {
        server.decrementServiceCounter();
        server.addResponse("TOPIC", response);
        System.out.println("Topic reserved");
      } else if (response.getResultCode() == ResultCodes.STATUS_ERROR ||
                 response.getResultCode() == ResultCodes.STATUS_INVALID ||
                 response.getResultCode() == -1) {
        server.decrementServiceCounter();
        server.addResponse("TOPIC", response);
        System.out.println("Topic error");
      } else if (response.getResultCode() == ResultCodes.STATUS_OK) {
        System.out.println("Topic created");
      }
    }
  } catch (Exception e) {
    System.out.println("Error " + e.getMessage());
  }
}
```

A service counter is used to control when all called services has replied. Each response is added to a map that also checks the counter, If all services has replied a confirm method is called.

```
public void incrementServiceCounter() {
  serviceCounter++;
  System.out.println("Increment: service counter: " + serviceCounter);
}

public void decrementServiceCounter() {
  serviceCounter--;
  System.out.println("Decrement: service counter: " + serviceCounter);
}

public void addResponse(String service, ResponseEvent response) {
  try {
    responseList.put(service, response);

    if (0 == serviceCounter) {
      confirm();
    }

  } catch (Exception e) {
    System.out.println("Error in addResponse: " + e.getMessage());
  }
}
```

The confirm method sends a confirmation to all services in the map, if all of them got a reservation. If not, the method returns and no confirmation events are sent. Unconfirmed reservations are removed during automatic reservation cleanup.

```
private void confirm() throws ArgumentOutOfRangeException, SystemException {

  if(getServiceStatus().equals("ERROR")) {
    return;
  }

  for (Map.Entry<String, ResponseEvent> entry : responseList.entrySet()) {
    ConfirmationEvent confirmation = null;

    if (entry.getKey().equals("TOPIC")) {
      confirmation =
                EventMessageBuilder.buildConfirmationEvent(entry.getValue().getKey(),
                               KSSServerBusiness.TOPIC_RESPONSE_QUEUE, "KSS_SERVER");

      getTopic().getProducer().publish(KSSServerBusiness.TOPIC_REQUEST_QUEUE,
                                                   confirmation.getReserveKey(),
                            new GsonUtils<ConfirmationEvent>().toJSON(confirmation));
    } else if (entry.getKey().equals("EVENT")) {
      …
  }

  responseList.clear();
  serviceCounter = 0;
}
```

In short:

- The reservation pattern handles distributed transactions across services by using a two-pass protocol. First, the caller makes a reservation, if everything is valid; "OK" is returned together with a reservation key and an expiration time. Later an acknowledgement is send from the client that the requested operation should be executed.
- Using the reservation pattern to handle distributed transactions ensures consistency across services.

4.2.4. Microservice framework & architecture

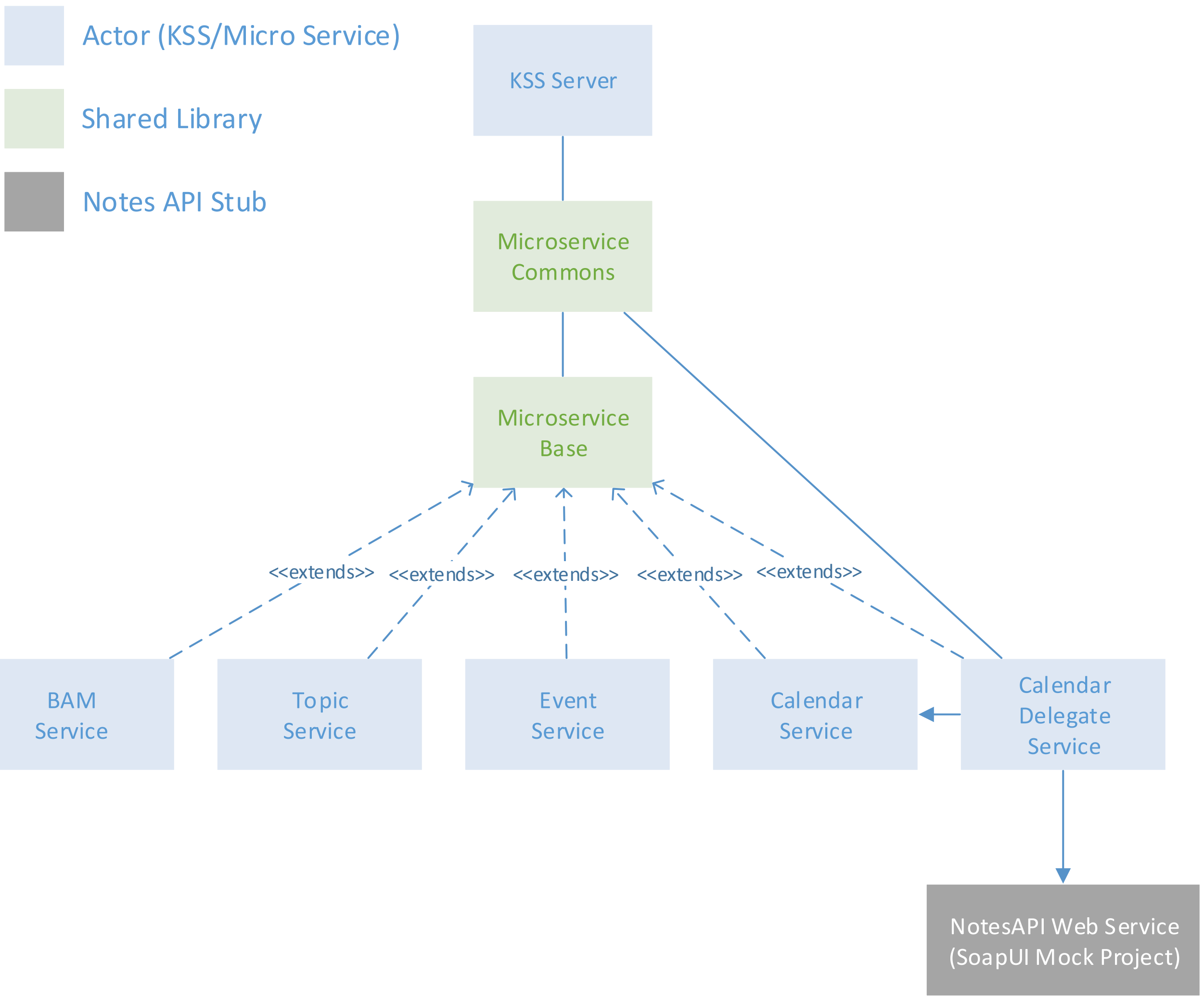


Figure 4.4 KSS Server microservice architecture.

My overall goal during system design was to implement a microservice framework that could hide all basic messaging and handshake functionality. This would simplify and ease the implementation of the final microservices and their interactions.

The KSS server component expose the same functionality as the existing component in the monolith version. It does however not hold any functionality itself, but is purely an orchestration component. It communicates with the microservices through the Microservice Commons Library described next.

To expose the NotesAPI interface a SoapUI mock project is introduced. It does not contain a version of the real interface, but works as a simple front to mimic the real world integration setup.

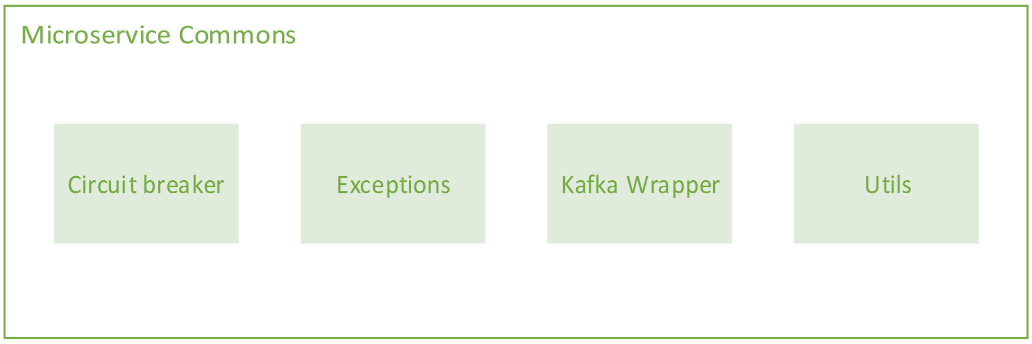


Figure 4.5 Microservice Commons component library architecture.

The Microservice Commons Library expose functionality for secure integration with the Kafka Message Queue server, JSON parsing and a circuit breaker component used to shield a caller when making remote calls, or elsewhere the component makes sense.

Circuit breaker
The generic circuit breaker component developed during the first iteration is added as part of the framework, specifically designed for secure integrations and elsewhere throughout the application where its usage makes sense.

Exceptions
Two application specific exceptions Business- and System Exception are identified and used throughout the system. The Business Exception is for validations or other expected failures. The System Exception is used as an encapsulation, if the system throws an unexpected exception like a null pointer.

Kafka wrapper
Contains both a Kafka producer and consumer for publishing and receiving messages on Kafka queues. The Kafka wrapper uses the circuit breaker component to shield the framework caller from calling the Kafka API directly, when a specified failure count is reached. Another messaging provider, like RabbitMQ, can easy replace this wrapper without affecting any clients that use this library.

Utils
This package contains functionality for parsing Java objects to and from JSON, using Google’s GSON library.

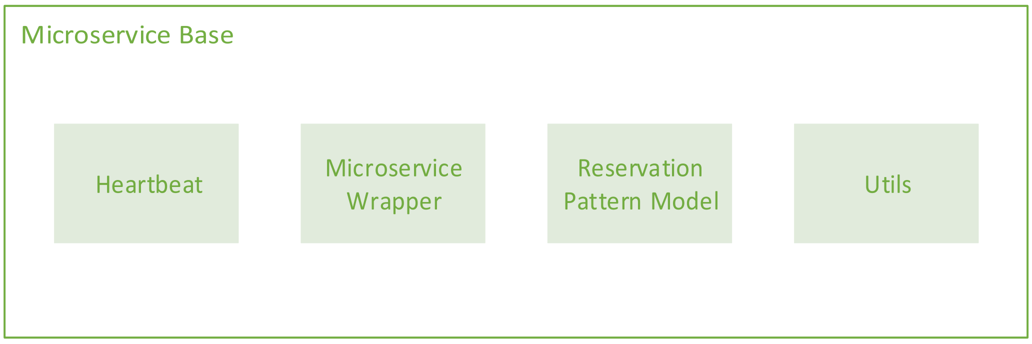


Figure 4.6 Microservice Base component library architecture.

The Microservice Base library contains all basic functionality needed when exposing a Microservice.

Heartbeat/Handshaking

A microservice system might eventually consist of a large number of services. To keep track of the state of each service a heartbeat or "is alive" functionality is implemented. The heartbeat functionality sends a "ping" every 5 seconds to a surveillance queue. The queue is read by an existing monitoring application at Alm Brand, which shows a dashboard of the current state of each service or application currently in production.

In a decoupled system, it is highly likely that a service might not respond or be slow from time to time. In situations where services does not respond, there is no reason for calling them. It is a waste of CPU cycles and results in slow response time waiting for potentially nothing, giving the user or caller a poor experience.

A way to prevent this scenario is to implement a heartbeat or ping/echo tactic suggested by Nygard [23, Chapter 5] throughout the systems integration points.

Using a heartbeat means that the microservice regularly sends a notification to somewhere else, as oppose to getting a request for one, which is commonly known as a ping/echo.

Either solution would make sense in this setup; using the ping/echo in the architecture would support a setup where a service client could request for a ping/echo, while a heartbeat is typically aimed at a specific receiver like a surveillance component. The heartbeat solution offer better scaling because it only sends out one heartbeat, and lets the queue provider fan out the heartbeat to all clients listening on the heartbeat queue. This setup uses a pre-defined heartbeat queue, shown later.

Each microservice has its own heartbeat thread running, located within the microservice base library. It sends out a "ping" message every 5 seconds, which ideally should be part of the service configuration on how often the ping is sent. If no ping is received, it goes without saying, means that the service is down.

By implementing the heartbeat tactic, the system supports a general requirement from Alm Band of having a monitoring component of the entire system, making surveillance simple and convenient to the administration department.

Going forward the "ping" message might be too simple, and should ideally contain more information. Each microservice could for instance offer information about its database state, circuit breaker state, failure and request count and so on. This information would then be available for dashboards showing each service's up/down status and a variety of operational and business relevant metrics.

A simple timer task has been implemented that sends a ResponseEvent containing "ping" in the content object to the predefined surveillance topic.

```
class HeartBeatTimer extends TimerTask {
  @Override
  public void run() {
    if(surveillanceTopic != null && !surveillanceTopic.trim().equals("")) {
      ResponseEvent event = new ResponseEvent();
      event.setObject(PING);

      facade.getProducer().publish(surveillanceTopic, event.getKey(),
                                   new GsonUtils<ResponseEvent>().toJSON(event));
    }
  }
}
```

In short:

- A heartbeat is a periodic message exchange between the system monitor and the microservice.
- Implementing a heartbeat allows the system administrators for better fault detection.
- Quicker fault detection minimizes the amount of time a given service is down thereby improving the overall system stability/availability.
- Adding service state information to the heartbeat message adds support for service state dashboards. For instance showing operational and business relevant metrics.
- Implementing a range of service state information into the heartbeat and display those in dashboards around the offices can highlight problems, allowing administrators to act quickly before problems result in system failure. This will also help improve the systems stability/availability.
- Adding more clients using the ping/echo tactic, results in more requests, adding extra load on the server. The heartbeat tactic keeps the load down and uses the message queue provider to handle the load and fan out the heartbeat message to the clients.
- Adding the heartbeat tactic to the framework means that all microservices, both old and new automatically implements this functionality, ensuring a minimal amount of maintenance are required.

Microservice wrapper

Exposes an interface implemented by each microservice to gain access to the basic messaging and reservation pattern functionality.

A timer task is added to mark and delete all expired reservations. A reservation is marked as expired after a customized period and deleted after 24 hours. The expired period is solely for logging and notification purposes, it makes no difference if a reservation has expired or does not exist.

Reservation pattern model

The Reservation pattern model consists of three event classes. RequestEvent, ResponseEvent and ConfirmationEvent as shown in Figure 4.3 Reservation model. The RequestEvent and ConfirmationEvent is send through the interface to the end microservice implementation. Reservations are kept in memory and handled by the microservice wrapper implementation.

Utils

Contains functionality to build response events and holds predefined status codes.

All the microservices in this project and the KSS Server component share the same basic architectural layout, as shown in figure 4.7.

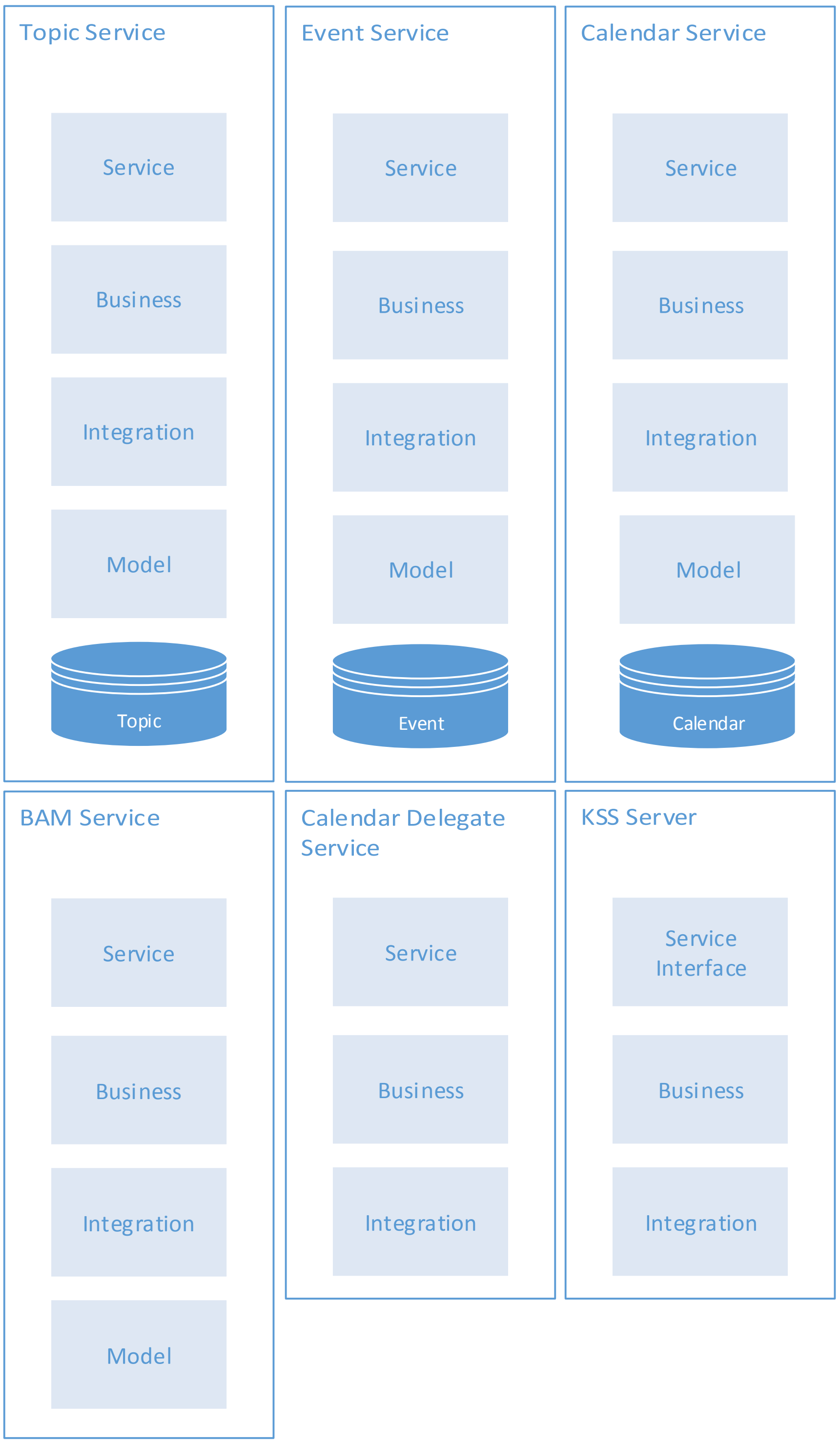


Figure 4.7 Microservice & KSS Server component architecture.

Service

The service layer contains the actual service implementation, which extends from the microservice base library. At the same time, it is also responsible for object mapping between the interface model and the service model when this is required.

Business

Is the actual business logic of the service. E.g., the calendar service validates incoming reservation requests. Is the time slot free? - Is there currently a reservation occupying this timeslot, is the end date after the start date etc. All the services has fields that are required, validations are done to make sure these fields are not empty etc.

Integration

This layer has the facade implementation for whatever integration the service requires. E.g. database, jms queue or web service.

Model

Contains the model classes or POJO's in the service. Could be an Event, Topic, Meeting etc.

Database

The Topic-, Event- and Calendar services needs to persist data, and therefore has their own isolated data storages, which is 100% decoupled from the other services involved in this use case.

4.2.5. Microservice reservation protocol

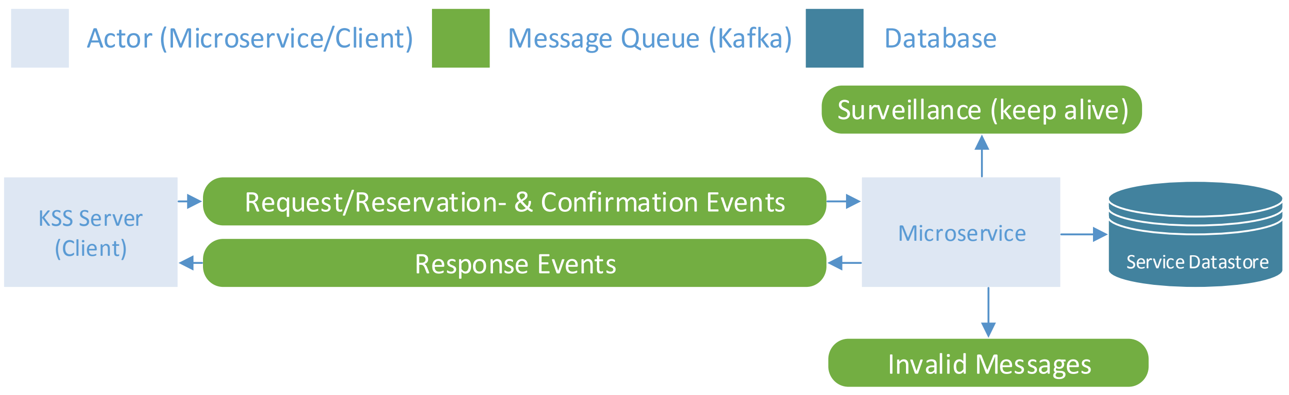


Figure 4.8 Microservice messaging flow.

Figure 4.8 illustrates the message flow of a microservice. I have based the messaging interface found in the microservice base library on the reservation pattern described in section 2.2.3 Reservation Pattern and the reservation events identified and implemented in section 4.2.3 Distributed Transactions.

As mentioned all services in this system communicates using JSON based messages and a custom protocol that consist of three basic event message formats. A request/reservation-, confirmation and response message.

Request/Reservation

Time:	1048261736520 ms
Key:	Uuid
Operation:	CreteOrUpdateMeeting

AckNeeded:	1
Reply to:	event_response_queue (Kafka Topic)
Contents:	message (JSON)

The request or reservation event is the first message in the flow. If the request is an update/delete or otherwise request to manipulate existing data a reservation is performed and a reservation id is returned (The id is the same as the uuid key). If the request is to retrieve some data, for instance a getAllMeetings operation, the result is returned, no reservation is performed and therefore no confirmation is required.

Confirmation

Time:	1048261736520 ms
Key:	Uuid
AckNeeded:	1
Reply to:	calendar_response_queue (Kafka Topic)
Confirmation code:	1

The confirmation event is send by the client to confirm a reservation. No content is needed, but only the reservation key and a confirmation code. If the reservation is not found or has expired an error is returned to the client. When a client sends a confirmation the data is validated a second time, just in case before inserted or added to the service data storage.

Response

Time:	1048261736520 ms
Key:	Uuid
Result code:	1
Result description:	OK
Contents:	message (JSON)

All responses from a service has the same layout. If a retrieve data request is received, a response is returned with the result together with a result code notifying the client if the request has succeeded or failed. If it is a reservation, the reservation number and the result code 200 = Reservation_OK is returned.

4.2.6. Topic microservice sequence diagram

The Topic service from the first iteration is changed to implement the framework library and protocol. All microservices has the same basic layout and call sequence. The Topic service sequence diagram show what flow the services go through when they receive a request and confirmation message.

Because it would take a lot of time to include everything and the benefit would be limited, the services in this project only contains a subset of the attributes that exists in the existing monolith version.

The sequence diagram is found on the next page.

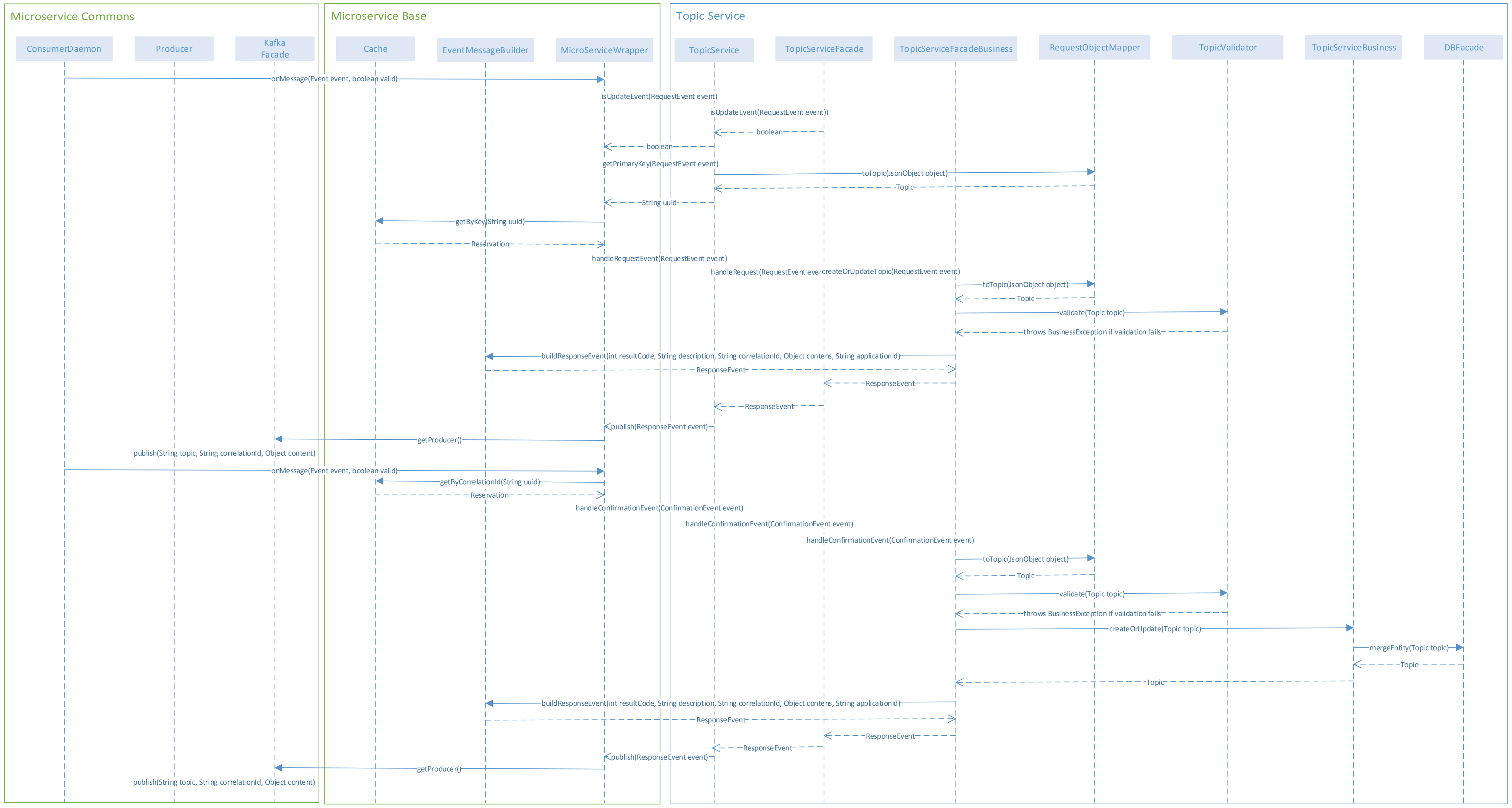
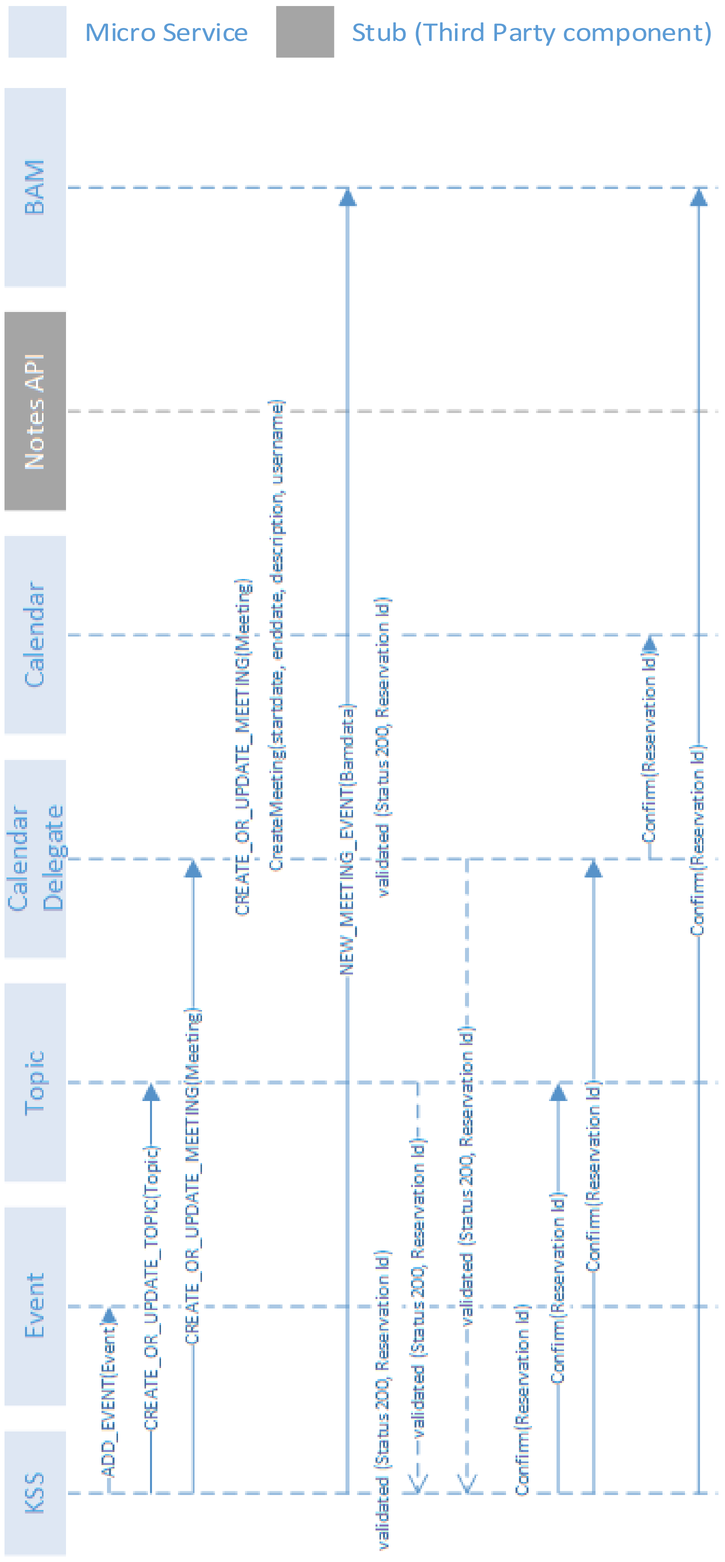

Microservice Commons
Microservice Base
Topic Service
ConsumerDaemon
Producer
Kafka Facade
Cache
EventMessageBuilder
MicroServiceWrapper
TopicService
TopicServiceFacade
TopicServiceFacadeBusiness
RequestObjectMapper
TopicValidator
TopicServiceBusiness
DBFacade
onMessage(Event event, boolean valid)
isUpdateEvent(RequestEvent event)
isUpdateEvent(RequestEvent event))
boolean
boolean
getPrimaryKey(RequestEvent event)
toTopic(JsonObject object)
Topic
String uuid
getByKey(String uuid)
Reservation
handleRequestEvent(RequestEvent event)
handleRequest(RequestEvent ever
createOrUpdateTopic(RequestEvent event)
toTopic(JsonObject object)
Topic
validate(Topic topic)
throws BusinessException if validation fails
buildResponseEvent(int resultCode, String description, String correlationId, Object contens, String applicationId)
ResponseEvent
ResponseEvent
ResponseEvent
publish(ResponseEvent event)
getProducer()
publish(String topic, String correlationId, Object content)
onMessage(Event event, boolean valid)
getByCorrelationId(String uuid)
Reservation
handleConfirmationEvent(ConfirmationEvent event)
handleConfirmationEvent(ConfirmationEvent event)
handleConfirmationEvent(ConfirmationEvent event)
toTopic(JsonObject object)
Topic
validate(Topic topic)
throws BusinessException if validation fails
createOrUpdate(Topic topic)
mergeEntity(Topic topic)
Topic
Topic
buildResponseEvent(int resultCode, String description, String correlationId, Object contens, String applicationId)
ResponseEvent
ResponseEvent
ResponseEvent
publish(ResponseEvent event)
getProducer()
publish(String topic, String correlationId, Object content)

Figure 4.9 Topic microservice interaction diagram

Use case sequence diagram

The use case diagram below shows the overall call sequence for the selected use case “Create meeting from Absalon”.



Micro Service
Stub (Third Party component)
BAM
Notes API
Calendar
Calendar Delegate
Topic
Event
KSS
ADD_EVENT(Event)
CREATE_OR_UPDATE_TOPIC(Topic)
CREATE_OR_UPDATE_MEETING(Meeting)
CREATE_OR_UPDATE_MEETING(Meeting)
CreateMeeting(startdate, enddate, description, username)
NEW_MEETING_EVENT(Bamdata)
validated (Status 200, Reservation Id)
validated (Status 200, Reservation Id)
validated (Status 200, Reservation Id)
validated (Status 200, Reservation Id)
Confirm(Reservation Id)
Confirm(Reservation Id)
Confirm(Reservation Id)
Confirm(Reservation Id)
Confirm(Reservation Id)

Figure 4.10 Create Meeting use case diagram

4.2.7. Partial conclusion

A goal of the prototype was partly to build a minor working version of the existing use case “Create meeting from Absalon” and throughout this process build a generic microservice framework and protocol that is applicable as-is at Alm Brand.

The microservice framework is still at an early stage, but has shown great potential. It takes about 5 minutes to get a microservice stub up and running using the library. This means that developers can focus on the business functionality and not worry about the rather large amount of overhead that follows, handling distributed transactions, Kafka communication, and handshaking.

I developed a simple protocol as part of the framework based on the Reservation pattern. It was straight forward to implement but resulted in a chatty protocol that currently does not offer an option to cancel a reservation. Instead this is handled using the built in cleanup functionality.

The prototype only consist of that one use case; this however lays the foundation for five different microservices and two working mock services, which interact with one another and the overall KSS server component, in three different ways. All the microservices interact through a number of Kafka request/reply queues, while the mocks expose their functionality as a web service using a SOAPUI mock project or through RabbitMQ request/reply queues.

The responsibilities and scope of each microservice was identified through meetings with stakeholders at Alm Brand, and code reviews of the existing application. After that, each of them was designed based on their corresponding business capabilities and common microservice rules and characteristics.

Splitting a system into a large amount of smaller self-contained services, makes it hard and time consuming to monitor and keep track of the state of each service. In order to address this concern a handshaking functionality is introduced into the generic microservice framework.

In order for a service to follow microservice guidelines, each service must have its own log. During implementation, another concern emerged. If the services are located at different servers and these servers time definitions are not identical, it can complicate troubleshooting. In that case, which service was called in what order?

5. Discussion

5.1. Discussion of the microservice paradigm

The microservice term is relatively new; most literature on the topic is found on web sites, blogs, forums and the likes. The term does currently not have a clear definition. In this chapter, I will provide my own understanding of the term and style based on the literature that got me here, and experiences doing the prototype.

5.1.1. Characteristics of a microservice revisited

There is a general lack of guidelines to microservices. Most agree that microservices are small and individually deployable. Some also believe a microservice must be between 10-100 lines of code, after striping all frameworks, third party libraries etc. Some take the size of the domain into account, while others also include the amount of people working on the service as part of their definition. However, I consider these characteristics or guidelines too vague and fluffy.

At the start of this thesis, I gave a list of microservice characteristics primarily based on reading Martin Fowler [1]. After developing the prototype I have narrowed the list down, to what I believe are four main criteria's a service must abide in order to characterize it as a microservice.

5.1.1.1. Focus on business capabilities

A typical development team includes UI designers/developers, backend developers, and database experts. Alm Brand and many other companies split the development process into several components, such as *UI*, *backend*, *database* and *mobile* applications. To make changes that will affect one of them, you need to agree these changes with other team members working on the project and maybe even get a budget approval from the business stakeholders. Using the microservice approach, the development process becomes more flexible. Each service is a separate application with its own business logic, UI, data store, and functionality for connecting with other services. Therefore, every member of the team may need experience in backend- and frontend development as well as database implementation. This approach enables developers to add new features quickly without risking the stability and operation of the entire system.

5.1.1.2. Independence of self-contained services

Each service contains its own business logic and is deployed separately. This makes it possible to change and gradually extend with new features without effecting the rest of the system. The self-contained property also support the flexibility to select the technology best suited for the specific service. Not being restricted to a predefined set of technologies or frameworks.

5.1.1.3. Decentralized data management

All services has its own data store, decoupling it completely from all other parts of the system. This will typically result in small and simple database schemas, which viewed separately, is great because it ensures an easy overview of the schema. The data will on the other hand also be decoupled, which may require the same data stored in different locations. This is risky and requires a lot of management and testing, to ensure that a system never updates or deletes the data in one service without updating or deleting the corresponding data in other services that contain the same data or reference set.

The selected use case did not have this issue, but going forward it is a concern that would need to be addressed.

As the data only has one owner, but potentially several update subscribers, a way to solve this is using a broadcast functionality. When a microservice is updated, it broadcasts the updated data to all services that listens on a specific broadcast queue. This retains the decoupled architecture, and at the same time makes it easy to extend with more services/clients that needs to listen for updates from a particular service. This is also known as an eventual consistency strategy. Eventual consistency means that, instead of requiring immediate consistency, the data in the system will eventually become consistent over time. Using eventually consistency requires a need to address the inconsistency periods that may occur.

5.1.1.4. Fault tolerance

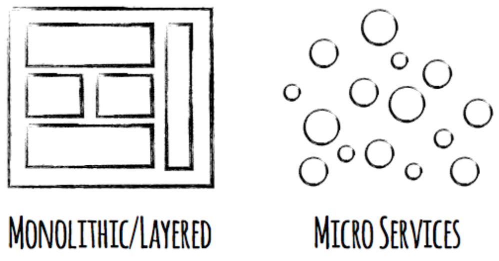
Microservice systems consist of multiple small units that can all fail; these units are loosely coupled and can be automatically restored. In addition, there are many ways for developers to increase availability of each separate unit, like for instance the circuit breaker and bulkheads mentioned in this thesis. This results in a more stable and robust system.

5.2. Microservices vs Monolith

Both microservice- and monolith systems can be categorized as SOA style systems. They base their architecture on services to achieve a loose coupling, which in theory makes each component easy to move, reuse, extend, maintain and replace.

Microservices takes the concept of loose coupling a step further.

In this section, I will provide a short discussion based on my own experiences with monolithic systems and the microservice prototype done in this thesis.



5.2.1. Development/Maintenance

Viewed isolated, microservices have great potential, they are small and self-contained, which makes them easier to develop, maintain and test. The issue comes when having a large number of services, which all needs their own build, test, deploy and run, potentially polyglot environments. They might also for a period have to support several versions, until all clients are prepared to use the new version. Meanwhile, do we know how many applications that actually use a particular microservice? – This needs to be registered; otherwise, things will go wrong during service interface updates. This is also a concern in traditional monolith systems, but the number of overall clients are likely to have multiplied in microservice style systems making it even more important.

Developing a monolith requires a larger team, and clear agreements and guideline on who does what. The monolith might be split into a number of modules. Nevertheless, it is still developed, maintained, tested and run as one large application. This is both a downside and an upside, the monolith is deployed as one, and small changes might affect several components and require a larger amount of regression testing. However, the team is in control, allowing them to decide when and how a change is done.

Meanwhile because of the size of a monolith often there is little desire or a lack of overview to refactor the system code, introducing a risk of technical debt. Because of its size, the technical debt is smaller and easier to address for microservices.

Like I mentioned in section 4.2.7. Part conclusion, issues regarding troubleshooting in a distributed environment is worth mentioning as well. As each microservice by default has its own log, and the services are typically located on different servers. When a use case involving several services fail, what happens if the involved servers' timestamps are out-of-sync? - Then there is no way of knowing which service was called when and in what order, making troubleshooting extremely difficult. In a monolithic system, all components run on the same server and share the same log, making troubleshooting a lot easier.

5.2.2. Operations overhead

Scaling monolith systems require a load balancer and a full monolith deployed to several servers. It is not possible to scale one part of a monolith; it is always everything or nothing. This is supported by microservices, as they are both small and self-contained, and communicates using a message queue, another instance of the service can simply be started, and listen on the same queue. On the other hand having potentially tens of separate services deployed across a large number of different servers, add messaging servers, database servers, failover and load balancers the size of the overall environment might very well exceed the size of a comparable monolith environment. Throw in the fact that a microservice system might be written in several different languages, and run on different types of application servers. For these reasons, choosing a microservice approach also requires highly skilled and versatile system administrators.

5.2.3. Complexity of distributed systems

Microservices introduce lots of remote procedure calls, REST APIs or messaging for these components to communicate together across different processes and servers. A distributed system of this size raises concerns regarding network latency, fault tolerance, message serialization, unreliable networks, synchronicity, versioning, varying loads within the application tiers etc.

Some of these can be solved through development; backwards compatibility is a property that might not be as required within the monolithic alternative.

Considerations regarding distributed systems are not needed to the same extend in the monolith alternative. The cost of this however is that the developer must handle all these considerations and risks, which he did not have to before. Distributed systems are an order of magnitude more difficult to develop and test against than the monolith system.

5.2.4. Testability challenges

When many services all evolve at different paces, it can be difficult to recreate consistent environments for either manual or automated testing. Adding synchronicity and dynamic message loads makes it harder to test microservice systems and have confidence in the set of services that are going into production.

As mentioned in section 5.2.1 Development/Maintenance, it is easy to test the individual microservice isolated, but in a dynamic environment, even small changes can affect the interactions of the services, which are hard to visualize and speculate on, let alone comprehensively test.

In a monolith system, this is often within the project teams´ control, making full system testing easier. However, testing smaller parts is likely more difficult than a microservice system.

6. Future Work

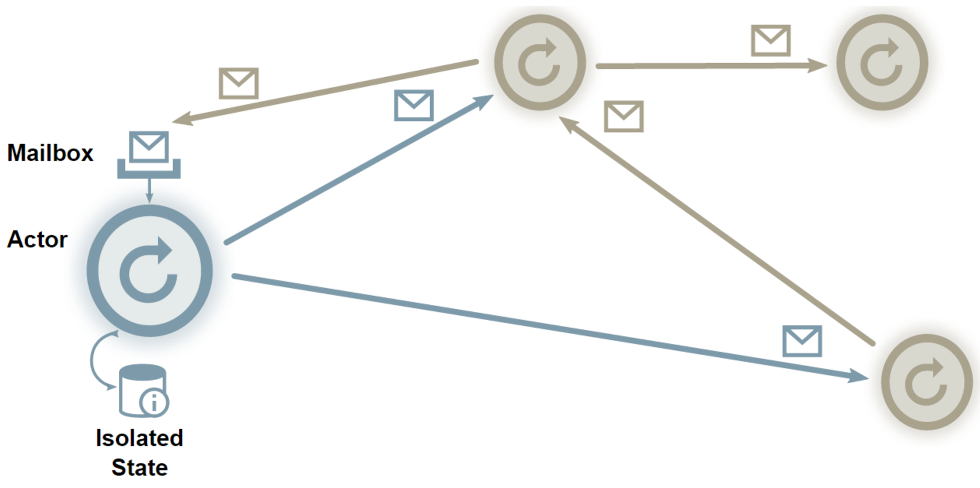
This section deals with what to do, and where to go from here. The prototype still needs work and some of the missing pieces that still needs to be addressed are described below.

Distributed Transactions

More candidates to handle distributed transactions should be investigated. One of these candidates, is the Akka Framework [26]. Akka offers a great deal of functionality for building concurrent and distributed messaging applications. Amongst them Akka expose a version of the actor model that might be a better fit than the reservation pattern implemented.

The Actor model provides a higher level of abstraction for writing concurrent and distributed systems. It relieves the developer from having to deal with explicit locking and thread management, making it easier to write concurrent and parallel systems.

Everything in Akka is designed to work in a distributed environment: all interactions between actors use pure message passing and everything is asynchronous.



6.1 Actor model

Each actor has its own mailbox, which is used for communication. The mailbox is read sequentially, and the actor can send messages to itself for later processing. All message passing within the actor model is asynchronous and each actor can have no, one or several addresses which maps to the same mailbox. Actors are isolated; so they do not share anything like state, resources e.g. this allows a single actor to fail without affecting any other actors. Furthermore, the actor model itself can be used for fault tolerance, by spawning hierarchy trees of actors for supervision. Once an actor crashes, the supervising actor receives a message and can react. A supervisor might restart the actor, stop other actors or escalate by sending an error message to its own supervisor.

Another candidate or strategy to solve the distributed transactions issue is to avoid them all together and instead relax the immediate consistency requirement and rely on the eventual consistency strategy mentioned in section 5.1.1.3, and introduce fallback strategies if something goes wrong, also used by eBay [29].

Testing

The microservice framework is at an early stage and the amount of testing it has undergone is limited. A full test analysis and test suite needs to be developed to ensure the framework is both robust and reliable, and at the same time performs as required, before actively used on a project.

7. Conclusion

The focus was to develop a partial version of the existing use case "Create meeting from Absalon", both to gain experience with the microservice architecture style, and research solutions to ensure a high level of availability as well as ways to reduce the amount of maintenance needed across the microservices through reuse.

The issues that I intended to investigate as defined and highlighted in the problem specification is included for reference.

The first:

> Investigate availability tactics for the KSS use case using a microservice architecture. Specifically I will experiment with circuit breaker and bulkhead implementations and compare the resulting availability with that of the existing KSS architecture.

Various patterns and solutions was considered to ensure high availability. I decided to implement a version of the circuit breaker pattern combined with timeouts suggested by both Nygard [23, chapter 5] and Fowler [24]. The circuit breaker pattern shields the application from external calls, this ensures a high level of protection from third party applications like Kafka. Resulting in a more robust system and thereby improving the overall system stability/availability.

Bulkheads was also introduced as a way to ensure high availability. Because of the architectural nature of a microservice system and its use of message queues for communication, the system automatically adopts the bulkhead pattern. The services are loosely coupled by nature, ensuring that an exception in one service does not affect the rest of the microservices within the system. Using Kafka as message broker also supports multiple instances of a service, which helps improve the systems overall availability/stability.

A heartbeat functionality was also implemented for better service state overview, as the number of microservices increase. The heartbeat was added to the microservice library shielding it from the service developer. This can also help to increase availability, as failures and service breakdowns can be caught early keeping potential downtime to a minimum.

There does not exist any metrics on the current monolith systems availability, so my intension of comparing the availability of the existing system to the microservice styles system was not possible. Nevertheless, the availability tactics implemented in this prototype will help achieve a high-level of availability and thereby contribute to a robust and reliable microservice system.

The second:

> Investigate modifiability tactics for the KSS use case, focusing on lowering maintainability costs through designing and implementing a generic microservice framework.

The development of a microservice commons library was built. This ensures that all shared functionality is isolated within one library, both minimizing the amount of testing needed, but also helps to keep the development maintenance costs down for any library modifications. When the commons library is changed, the changes are automatically materialized down through the services that implements the library, which will be all microservices within the system.

The microservice framework is still at an early stage but has shown great potential. It takes about 5 minutes to get a microservice stub up and running using the library. This means that developers can focus on the business functionality and not worry about the rather large amount of overhead that follows, handling distributed transactions, message communication, and handshaking.

I developed a simple protocol as part of the framework based on the Reservation pattern. It was strait forward to implement but resulted in a chatty protocol that currently does not offer an option to cancel a reservation. Instead this is handled using the built in cleanup functionality.

Alm Brand has an expectation that developing microservice systems is easier and simpler than doing the existing monolith systems. Based on this thesis I do not necessarily believe that to be true.

My framework showed that it is easy to get a microservice stub running, but there is a large amount of communication overhead, handling of distributed transactions, shared data across microservices and so on that needs to be addressed. The system complexity does not diminish using microservices; it is simply moved from one part of the system to another being from the service to the caller. Today all business functionality is typically done within the backend server. In a microservice system, a part of it is split into each service while the rest needs to be handled by the caller.

Microservices is an interesting architecture style. Once you have handled the overhead mentioned in this thesis, I believe there is no reason not to seriously consider the microservice style instead of the monolith style. Not saying it is better, but the saying "*what you lose on the swings, you gain on the roundabouts*", is well suited in this situation. It is a matter of what the end goal is. However, when considering microservices it is important not to get carried away by its current hype.

Buttom line - Choosing the microservice approach offers more flexibility in terms of scaling like mentioned in section 4.2.2. Operational overhead. It is easier to implement elasticity and automatically start an extra instance of a service, if the load requires it. Each service is smaller and self-contained, making them easier to maintain, develop and deploy with limited risk of breaking functionality elsewhere within the system. It provides the freedom to choose whatever technology is best suited for the specific service. Finally yet importantly, a major issue today is the fact that the entire monolith has to be tested and deployed to production, even for the smallest bug fixes. The entire build cycle for microservices are quicker and more in line with today's requirements of agility and many smaller releases.

8. References

[1] Martin Fowler and James Lewis. Microservices (Accessed on 03/02/2015).
URL: http://martinfowler.com/articles/microservices.html

[2] Illinois Institute of Technology (Accessed on 02/02/2015).
URL: http://omega.cs.iit.edu/~ipro329/methodology.html

[3] L. Bass, P. Clements & R. Kazman - Software Architecture in Practice (2nd Edition), 2003

[4] Paul A. Moore (Accessed on 03/02/2015)
URL: http://hosteddocs.ittoolbox.com/PM043004.pdf

[5] Grace Lewis (Accessed on 03/02/2015)
URL: http://www.sei.cmu.edu/library/assets/whitepapers/SOA_Terminology.pdf

[6] W3C (Accessed on 04/02/2015)
URL: http://www.w3.org/TR/ws-arch/#service_oriented_architecture

[7] Monolithic Design (Accessed on 04/02/2015)
URL: http://c2.com/cgi/wiki?MonolithicDesign

[8] Steve Jones blog (Accessed on 04/02/2015)
URL: http://service-architecture.blogspot.dk/2014/03/microservices-is-soa-for-those-who-know.html

[9] Sven Bernhardt. SOA Magazine 2014 (Accessed on 04/02/2015)
URL: http://www.opitz-consulting.com/fileadmin/redaktion/veroeffentlichungen/pdf/soa-magazine-2014-3_microservices-_architectures_bernhardt_sicher.pdf

[10] James Lewis blog (Accessed on 04/02/2015)
URL: http://www.thoughtworks.com/insights/blog/microservices-nutshell

[11] Apache Kafka: Next Generation Distributed Messaging System (Accessed on 22/02/2015)
URL: http://www.infoq.com/articles/apache-kafka

[12] Apache Kafka (Framework) (Accessed on 22/02/2015)
URL: http://kafka.apache.org/

[13] J. Kreps LinkedIn Corp, N. Narkhede LinkedIn Corp, J. Rao LinkedIn Corp.
Apache Kafka: a Distributed Messaging System for Log Processing (Accessed on 25/02/2015)
URL: http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf

[14] Apache Kafka & Zookeeper introduction (Accessed on 21/02/2015)
URL: http://www.slideshare.net/rahuldausa/introduction-to-kafka-and-zookeeper

[15] Apache Zookeeper (Accessed on 21/02/2015)
URL: http://zookeeper.apache.org/

[16] Clarifying the Saga Pattern (Accessed on 21/02/2015)
URL: http://kellabyte.com/2012/05/30/clarifying-the-saga-pattern/

[17] Saga Pattern (Accessed on 21/02/2015)
URL: http://www.rgoarchitects.com/Files/SOAPatterns/Saga.pdf

[18] Gianpaolo Carraro and Chong Fred. Software as a Service (SaaS): An Enterprise Perspective (Accessed on 20/02/2015). Tech. rep. 2006.
URL: http://msdn.microsoft.com/en-us/library/aa905332.aspx.

[19] Alain Abran, J W Moore, P Bourque, R Dupuis, and L L Tripp. "Software Engineering Body of Knowledge". In: IEEE Computer Society, Angela Burgess (2004).

[20] Ralph Mietzner, Tobias Unger, Robert Titze, and Frank Leymann. "Combining Different Multi-Tenancy Patterns in Service-Oriented Applications". In: (2009).

[21] Two-phase protocol (Accessed on 01/03/2015)
URL: http://en.wikipedia.org/wiki/Two-phase_commit_protocol

[22] Reservation Pattern (Accessed on 01/03/2015)
URL: http://www.drdobbs.com/windows/soa-patterns-reservation/228701670

[23] M. T. Nygard – Release It!, 2007

[24] Martin Fowler – Circuit Breaker Pattern (Accessed on 26/03/2015)
URL: http://martinfowler.com/bliki/CircuitBreaker.html

[25] The Saga pattern and that architecture (Accessed on 07/04/2015)
URL: http://arnon.me/2013/01/saga-pattern-architecture-design/

[26] Akka Framework of distributed transactions (Accessed on 25/04/2015)
URL: www.Akka.io

[27] Reservation Pattern (Accessed on 03/05/2015)
URL: http://www.rgoarchitects.com/Files/SOAPatterns/ReservationPattern.pdf

[28] Development/Maintenance ratio (Accessed on 03/05/2015)
URL: http://clarityincode.com/software-maintenance/

[29] Scalability Best Practices: Lessons from eBay (Accessed 08/06/2015)
URL: http://www.infoq.com/articles/ebay-scalability-best-practices

9. Appendix

9.1. Cache implementation

```java
public class Cache {

  private static Cache instance;
  private List<Reservation> reservationCache;

  private Cache() {}

  public static Cache getInstance() {
    if(instance == null) {
      instance = new Cache();
    }
    return instance;
  }

  private List<Reservation> getRequestEventCache() {
    if(reservationCache == null) {
      reservationCache = new ArrayList<Reservation>();
    }
    return reservationCache;
  }

  public void insert(Reservation reservation) {
    getRequestEventCache().add(reservation);
  }

  public void update(Reservation reservation) {
    for(Reservation reservation1 : getRequestEventCache()) {
      if(reservation1.getUuid().equals(reservation.getUuid())) {
        reservation1.setApplication(reservation.getApplication());
        reservation1.setData(reservation.getData());
        reservation1.setEntryDate(reservation.getEntryDate());
        reservation1.setPrimaryKey(reservation.getPrimaryKey());
      }
    }
  }

  public void remove(String uuid) {
    Reservation reservation = getByCorrelationId(uuid);

    if(reservation != null) {
      getRequestEventCache().remove(reservation);
    }
  }

  public Reservation getByCorrelationId(String uuid) {
    System.out.println("Looking up Uuid: " + uuid);

    for(Reservation reservation : getRequestEventCache()) {
      System.out.println("Current Uuid: " + reservation.getUuid());
      if(reservation.getUuid().equals(uuid)) {
        return reservation;
      }
    }
    return null;
  }

  public Reservation getByKey(String primaryKey) {

    if(primaryKey == null || primaryKey.equals("")) {
      System.out.println("getByKey primary key was empty or null");
      return null;
    }

    for(Reservation reservation : getRequestEventCache()) {
      if(reservation != null && reservation.getPrimaryKey() != null) {
        if(reservation.getPrimaryKey().equals(primaryKey)) {
          return reservation;
        }
      }
    }
    return null;
  }

  public List<Reservation> getAll() {
    return getRequestEventCache();
  }

  public List<Reservation> removeExpiredReservations() {
    List<Reservation> list = new ArrayList<Reservation>();

    for(Reservation reservation : getRequestEventCache()) {
      if(isMoreThan24HoursOld(reservation.getEntryDate())) {
        list.add(reservation);
      }
    }

    // Remove all expired reservations.
    getRequestEventCache().removeAll(list);
    return list;
  }

  private boolean isMoreThan24HoursOld(Date d) {
    long diff =  new Date().getTime() - d.getTime();
    return (diff / (24 * 60 * 60 * 1000)) > 1;
  }
}
```

9.2. MicroserviceWrapper.onMessage implementation

Below is the main microservice framework operation onMessage. It handles all incoming messages within the framework. Here the messages are analyzed to identity the message type. onMessage then calls the method handleEvent. HandleEvent takes care of reservation caching and verification as well as call the proper microservice operation handleRequestEvent, handleConfirmationEvent or handleResponseEvent implemented by each specific service as required by the MicroserviceWrapper interface.

```
@Override
public final void onMessage(String message) throws SystemException {
  JsonObject jsonObj = null;

  ConfirmationEvent confirmationEvent = null;
  ResponseEvent responseEvent = null;
  RequestEvent requestEvent = null;

  // Check for cast exceptions and pass the raw data back to the wrapper for logging
  // purposes.
  // Will happen if the message can't be passed to a JsonObject.
  try {
    jsonObj = new Gson().fromJson(message, JsonElement.class).getAsJsonObject();
  } catch (Exception e) {
    requestEvent = new RequestEvent(); // Dummy - Event is an abstract class, to one
                                       // of the others must do.
    requestEvent.setJson(message);
    handleEvent(requestEvent, false);
    return;
  }

  if (jsonObj.get("reserveKey") != null) {
    confirmationEvent = new GsonUtils<ConfirmationEvent>().fromJSON(message,
                                                            ConfirmationEvent.class);
    confirmationEvent.setJson(message);

    if (jsonObj.getAsJsonObject("object") != null) {
      confirmationEvent.setObject(jsonObj.getAsJsonObject("object"));
    }

    handleEvent(confirmationEvent, true);

  } else if (jsonObj.get("resultCode") != null) {
    responseEvent = new GsonUtils<ResponseEvent>().fromJSON(message,
                                                            ResponseEvent.class);
    responseEvent.setJson(message);

    if (jsonObj.getAsJsonObject("object") != null) {
      responseEvent.setObject(jsonObj.getAsJsonObject("object"));
    }

    handleEvent(responseEvent, true);
  } else if (jsonObj.get("responseQueue") != null) {
    requestEvent = new GsonUtils<RequestEvent>().fromJSON(message,
                                                                 RequestEvent.class);
    requestEvent.setJson(message);

    if (jsonObj.getAsJsonObject("object") != null) {
      requestEvent.setObject(jsonObj.getAsJsonObject("object"));
    }

    handleEvent(requestEvent, true);
  } else {
    logger.error("Unknown message: " + message);
  }
}

private void handleEvent(Event event, boolean valid) throws SystemException {
  try {
    // If its not a valid message, log the raw data to invalid queue.
    if (!valid) {
      logger.error("Invalid message");
      logInvalidMessage(event, event.getJson(), null, "Unknown");
      return;
    }

    if (event.getObject() instanceof String &&
                      String.valueOf(event.getObject()).equals(HeartbeatImpl.PING)) {
      heatbeat.setIsAlive(true);
      return;
    }

    // Aimed at the end implementation.
    if (event instanceof ConfirmationEvent) {
      ConfirmationEvent confirmation = (ConfirmationEvent)event;

      Reservation reservation =
                            Cache.getInstance().getByCorrelationId(event.getKey());
      if (reservation != null) {
        event.setObject(reservation.getData());
        handleConfirmationEvent(confirmation);
        Cache.getInstance().remove(event.getKey());
      } else {
        ResponseEvent response =
            buildResponseEvent(ResultCodes.STATUS_RESERVATION_NOT_FOUND, "Not able to
                                  find reservation or its expired", event.getKey(),
                                confirmation.getObject(), event.getApplicationId());
        publish(response, ((ConfirmationEvent)event).getResponseQueue());
        return;
      }
    } else if (event instanceof RequestEvent) {
      RequestEvent requestEvent = (RequestEvent)event;
      String primaryKey = "";

      if (isUpdateEvent(requestEvent)) {
        primaryKey = getPrimaryKey(event);
        if (Cache.getInstance().getByKey(primaryKey) != null) {
          ResponseEvent response = buildResponseEvent(ResultCodes.STATUS_RESERVED,
             "Data requested is already reserved", event.getKey(), event.getObject(),
                                                          event.getApplicationId());
          publish(response, requestEvent.getResponseQueue());
          return;
        }
      }

      int resultCode = handleRequestEvent(requestEvent);

      if (isUpdateEvent(requestEvent) && resultCode ==
                                        ResultCodes.STATUS_RESERVATION_OK) {
        // Message is a re-reservation. Update timestamp and return.
        Reservation reservation =
                        Cache.getInstance().getByCorrelationId(requestEvent.getKey());
        if (reservation != null) {
          reservation.setEntryDate(new Date());
          Cache.getInstance().update(reservation);
        } else {
          // If primary key is null or empty its a new object. No need to
          // reserve anything.
          Cache.getInstance().insert(new Reservation(event.getKey(), new
          Date(), event.getObject(), primaryKey, event.getApplicationId()));
        }
      }

    } else if (event instanceof ResponseEvent) {
      handleResponseEvent((ResponseEvent)event);
    }
  } catch (Exception e) {
    logger.error("Error message: " + e.getMessage(), e);
    logInvalidMessage(event, event.getJson(), null, "Unknown");
    return;
  }
}
```

9.3. Circuit breaker implementation

The circuit breaker component developed as part of the first iteration.

Package – dk.almbrand.microservice.common.circuitbreaker

The main package, which encapsulates the circuit breaker component.

CircuitBreaker

The interface that exposes the circuit breaker components functionality.

CircuitBreakerImpl

The main class that expose and handles the main circuit breaker states engine and timeout functionality. When the circuit breaker is initialized, the engine starts in the Closed state.

The class exposes two main methods to initiate the circuit breaker state engine. By offering the component functionality as either Callable or Runnable, the component offers flexibility for the developer to choose the most suited type for his/her needs, and at the same time a certain amount of decoupling from the rest of the library.

```
public <T> T doAction(Callable callable, Class<T> type)
public void doAction(Runnable runnable)
```

The doAction methods are called with an object that implements the Callable interface, in this case the ProtectedProducer that encapsulates the simple producer, which adds messages to a queue, using the Kafka API.

```
ProtectedProducerImpl protectedProducer = new ProtectedProducerImpl(producer, new
CircuitBreakerImpl(3, 6000 * 1000000));
```

```
public class ProtectedProducerImpl implements Callable, ProtectedProducer {
  …
  public void publish(String topic, String key, String message) {
    if(circuitBreaker != null) {
      //Temporary storing of values for when call is executed from the circuitbreaker
      this.topic = topic;
      this.key = key;
      this.message = message;

      circuitBreaker.doAction(this, Object.class);
    }
  }

  @Override
  public Object call() throws Exception {
    System.out.println("Sending message: "+message+" Topic: "+topic+" Key: "+key);
    producer.publish(topic, key, message);
    return null;
  }
}
```

One of the main benefits of choosing Callable is that the circuit breaker can catch any exception thrown inside the executing Callable method, which supports the circuit breaker failure count functionality. To catch exceptions thrown in a Runnable process it is necessary to implement a callback, which could compromise the decoupled architecture I am striving for.

To implement the lock functionality I am using a ReentrantLock, it supports timeouts while waiting for a lock; this ensures that the caller does not wait for a response longer than specified.

The method tries to execute the Callable until it either succeeds or reach the threshold marker. In this case the threadExecuter helps with the timeout functionality, by specifying the timeout in the call to get. The method protectedCodeIsAboutToBeCalled is only active in the Open state, which will result in an exception.

In case the state is HalfOpen the method protectedCodeHasBeenCalled changes the state back to Closed. In both the Closed and Open states, the operation does nothing.

When the request has been processed, the thread is shutdown.

```java
public <T> T doAction(Callable callable, Class<T> type) {
  boolean run = true;
  String result = "fault";
  logger.info("state is " + state.getClass());

  while (run && !thresholdReached()) {
    run = false;
    Future<String> futureResult = null;

    try {
      if (lock.tryLock(timeoutNanoSeconds, TimeUnit.NANOSECONDS)) {
        state.protectedCodeIsAboutToBeCalled();

        if (threadExecutor == null || threadExecutor.isTerminated())
          threadExecutor = Executors.newSingleThreadScheduledExecutor();
          futureResult = threadExecutor.submit(callable);

          result = futureResult.get(timeoutNanoSeconds, TimeUnit.NANOSECONDS);

          if (futureResult.isCancelled())
            throw new TimeoutException("Timeout marker reached");
      }

      if (lock.isHeldByCurrentThread())
        state.protectedCodeHasBeenCalled();

    } catch (Exception e) {
      logger.info("Exception, current state is " + state.getClass(), e);
      state.exceptionOccured(e);
      run = true;
    } finally {
      if (lock.isHeldByCurrentThread())
        lock.unlock();

      if (futureResult != null)
        futureResult.cancel(false);

      if (threadExecutor != null && !threadExecutor.isShutdown())
        threadExecutor.shutdown();
    }
  }
  return type.cast(result);
}
```

This method offers the same functionality as the previous only instead it takes a Runnable as input. The biggest difference is the timeout implementation is a little different. A downside as opposed to Callable, is that it is not possible to offer a return value from a Runnable, so that must be handled inside the client code.

In this scenario, the timeout functionality is not handed to me, but must be implemented manually. This is done using a while loop that keeps checking if the thread is alive. If this is the case, the time elapsed since its initiation is checked. In case it has passed the timeout marker, an exception is thrown and the thread is closed.

```
public void doAction(Runnable runnable) {
  Thread thread = null;
  boolean run = true;
  long timeout = 0;

  while (run && !thresholdReached()) {
    run = false;
    try {

      if (lock.tryLock(timeoutNanoSeconds, TimeUnit.NANOSECONDS)) {
        state.protectedCodeIsAboutToBeCalled();
        timeout = System.nanoTime() + timeoutNanoSeconds;
        thread = new Thread(runnable);
        thread.start();

        while (thread.isAlive()) {
          if (System.nanoTime() < timeout)
            throw new TimeoutException("Timeout marker reached");
        }
     }

     if (lock.isHeldByCurrentThread())
       state.protectedCodeHasBeenCalled();

    } catch (Exception e) {
      state.exceptionOccured(e);

      if (thread != null && !thread.isInterrupted())
        thread.interrupt();
        run = true;
    } finally {
      if (lock.isHeldByCurrentThread())
        lock.unlock();
    }
  }
}
```

Package - dk.almbrand.microservice.common.circuitbreaker.states

Contains the circuit breaker state classes.

CircuitBreakerState

Abstract class that specifies the state classes main functionality and holds the CurcuitBreaker handler object.

```
public abstract class CircuitBreakerState {
  protected final CircuitBreaker circuitBreaker;
  protected CircuitBreakerState(CircuitBreaker circuitBreaker) {
    this.circuitBreaker = circuitBreaker;
  }

  public void protectedCodeIsAboutToBeCalled() throws OpenCircuitException {}
  public void protectedCodeHasBeenCalled() {}
  public void exceptionOccurred(Exception ex) {
    circuitBreaker.increaseFailureCount();
  }
}
```

Closed

The initial state. When the engine goes into the closed state, the failure counter is reset to zero. If an exception is encountered in the Closed state, the failure counter gets incremented by one. When the threshold marker is reached the engine changes to the Open state.

```
public class Closed extends CircuitBreakerState {
  public Closed(CircuitBreaker circuitBreaker)  {
    super(circuitBreaker);
    circuitBreaker.resetFailureCount();
  }

  @Override
  public void exceptionOccurred(Exception e) {
    super.exceptionOccurred(e);

    if(circuitBreaker.thresholdReached())
      circuitBreaker.moveToOpenState();
   }
}
```

Open

Handles the Open state in the circuit breaker state engine. When Open is initialized, a timer is started and runs for the period specified during the circuit breaker component initialization.
When the timer runs out the state engine changes to the HalfOpen state.
In case the protected code is executed during the Open state, the OpenCircuitException is thrown.

```
public class Open extends CircuitBreakerState {

  public Open(CircuitBreaker circuitBreaker) {
    super(circuitBreaker);
    new Timer().schedule(new MoveToHalfOpenStateTask(),
    circuitBreaker.getTimeoutNanoSeconds());
  }
  @Override
  public void protectedCodeIsAboutToBeCalled() throws OpenCircuitException {
    super.protectedCodeHasBeenCalled();
    throw new OpenCircuitException();
  }

  class MoveToHalfOpenStateTask extends TimerTask {
    public void run() {
      circuitBreaker.moveToHalfOpenState();
      circuitBreaker.prepareFailureCountForHalfOpen();
      Thread.currentThread().~~stop~~();
    }
  }
}
```

HalfOpen

The engine moves into the HalfOpen state when the timer initialized in the Open state expires. The HalfOpen state is a temporary state and the next request determines if the state changes to either Closed or back to the Open state based on the success of the call.

```
public class HalfOpen extends CircuitBreakerState {
  public HalfOpen(CircuitBreaker circuitBreaker) {
    super(circuitBreaker);
  }

  @Override
  public void exceptionOccurred(Exception e) {
    super.exceptionOccurred(e);
    circuitBreaker.moveToOpenState();
  }


 @Override
  public void protectedCodeHasBeenCalled() {
    super.protectedCodeHasBeenCalled();
    circuitBreaker.moveToClosedState();
  }
}
```

Package - dk.almbrand.microservice.common.circuitbreaker.exceptions

Contains specific component exceptions.

ArgumentOutOfRangeException

The CircuitBreakerImpl constructor requires two parameters when a CircuitBreaker object is created. A threshold and a timeout, and both has to be greater than zero, is that not the case an ArgumentOutOfRangeException is thrown.

OpenCircuitException

The OpenCircuitException is thrown in the OpenState class, if the circuit breaker is in an open state when the method doAction is executed. This is simply to notify the caller that no calls are currently accepted.

Going forward a “nice to have” feature could be to return the remaining time that the engine has left before moving into the HalfOpen state to the client as part of the exception.

9.4. Topic.create_or_update_topic message flow

Below is an example of a full message flow for the create_or_update_topic operation based on the reservation protocol implemented as part of this thesis.

Step 1 – Client initialization

The message send from the client KSS Server requesting to create a topic:

```
{
  "ackNeeded": false,
  "operation": "CREATE_OR_UPDATE_TOPIC",
  "responseQueue": "topic_response_queue",
  "applicationId": "KSS_Server",
  "key": "9c4788cd-479a-411a-81fd-9cb559cfef30",
  "object": {
    "ab_customer_from": "May 29, 2015 2:48:58 PM",
    "ab_customer_to": "May 29, 2015 2:48:58 PM",
    "comments": "Han har ikke raad",
    "created": "May 29, 2015 2:48:58 PM",
    "creator": "ex0319",
    "customer_at": "Tryg Forsikring",
    "customer_at_startdate": "May 29, 2015 2:48:58 PM",
    "email": "net@net.net",
    "name": "Claus D. Nielsen",
    "no_of_adults": 2,
    "no_of_children": 0,
    "phone": "+45 51293953",
    "rki": 0,
    "senior": 0,
    "topic": "topic"
  }
}
```

Step 2 - Service reservation response

The reservation confirmation sent from the service as an acknowledgement.

```
{
  "resultCode": 201,
  "desc": "OK",
  "applicationId": "TOPIC_SERVICE",
  "key": "9c4788cd-479a-411a-81fd-9cb559cfef30"
}
```

Step 3 - Client reservation confirmation

Confirmation sent from the client to confirm the reservation.

```
{
  "reserveKey": "9c4788cd-479a-411a-81fd-9cb559cfef30",
  "ackNeeded": false,
  "operation": "",
  "responseQueue": "topic_response_queue",
  "applicationId": "KSS_SERVER",
  "key": "9c4788cd-479a-411a-81fd-9cb559cfef30",
  "object": {}
}
```

Step 4 - Service operation confirmation

Final confirmation from the service to the client that the topic is created. A copy of the newly created topic is included as part of the confirmation.

```
{
  "resultCode": 200,
  "desc": "OK",
  "applicationId": "TOPIC_SERVICE",
  "key": "9c4788cd-479a-411a-81fd-9cb559cfef30",
  "object": {
    "uuid": "ff2e655c-b9cf-4c4a-b5b4-4b763bba2e31",
    "ab_customer_from": "May 29, 2015 2:48:58 PM",
    "ab_customer_to": "May 29, 2015 2:48:58 PM",
    "comments": "Han har ikke raad",
    "created": "May 29, 2015 2:48:58 PM",
    "creator": "ex0319",
    "customer_at": "Tryg Forsikring",
    "customer_at_startdate": "May 29, 2015 2:48:58 PM",
    "email": "net@net.net",
    "name": "Claus D. Nielsen",
    "no_of_adults": 2,
    "no_of_children": 0,
    "phone": "+45 51293953",
    "rki": 0,
    "senior": 0,
    "topic": "topic"
  }
```

9.5. Environment configurations and installation guide

Below you can find descriptions and configurations needed to get the prototype running.

Link to Virtual Machine (Kafka): http://users-cs.au.dk/baerbak/c/vm/claus_nielsen_master_vm.zip

Link source code, configurations, mocks + logs: http://users-cs.au.dk/baerbak/c/vm/cn_ms_prototype.zip

Oracle Development Tools

This prototype has been developed using- and requires Oracle's JDeveloper environment in order to run.

Download and install JDeveloper: http://www.oracle.com/technetwork/developer-tools/jdev/downloads/jdev12120download-2220222.html

Download & install latest version of Maven: https://maven.apache.org/download.cgi?Preferred=ftp://mirror.reverse.net/pub/apache/

Import the projects into JDeveloper. (found in the zip file listed above)

Kafka / Virtual Machine

Download the virtual machine, which contains an installation of Kafka.

URL: http://users-cs.au.dk/baerbak/c/vm/claus_nielsen_master_vm.zip

Username/Passwords

The Ubuntu virtual machine credentials: Claus D. Nielsen/Herkules1

The virtual machine hostname is: ubuntu

Kafka

su root/Herkules1

cd /opt/kafka/kafka_2.8.0-0.8.1.1/bin/

Start Kafka: ./zookeeper-server-start.sh ./../config/zookeeper.properties & ./kafka-server-start.sh ./../config/server.properties

Stop Kafka: ./zookeeper-server-stop.sh

RabbitMQ

Download https://www.rabbitmq.com/download.html

Start RabbitMQ

SoapUI

Download SoapUI: http://www.soapui.org/getting-started/installing-soapui/installing-on-windows.html

Open project: /soapui/NotesAPI-soapui-project.xml (found in the zip file alongside the source code)

Start Mockserver on port: 8088

Database

Install MySQL: http://dev.mysql.com/downloads/windows/

Execute the following create scripts for each of the following services.

Topic: /database/topic_db/create.sql (found in the zip file alongside the source code)

Event: /database/event_db/create.sql (found in the zip file alongside the source code)

Calendar: /database/calendar_db/create.sql (found in the zip file alongside the source code)

The default database credential configuration is localhost:3306 root/root.

In case this does not match your setup, it can be changed in persistance.xml, found in the KSS project Service/META-INF/persistance.xml

Run use case

An initiation class dk.almbrand.server.Initializer.java can be found in KSS, which initiates and executes the entire use case.