**Research report – individual project**

**(Microservice interactions)**



Yordan Doykov

Contents

[Problem 2](#_Toc160269168)

[Opportunity 2](#_Toc160269169)

[Main research question: 2](#_Toc160269170)

[Q1. What are the most popular ways to decouple microservices? 2](#_Toc160269171)

[Source 1: Good and bad practices 2](#_Toc160269172)

[Source 2: Practical work examples 3](#_Toc160269173)

[Conclusion 5](#_Toc160269174)

[Q2: What is a message broker 5](#_Toc160269175)

[Source 1: Library research 5](#_Toc160269176)

[Source 2: exploring RabbitMQ’s documentation 5](#_Toc160269177)

[Concluson: 6](#_Toc160269178)

[Q3: How does a message broker guarantee that messages are delivered? 6](#_Toc160269179)

[Source 1: Expert interview 6](#_Toc160269180)

[Source 1: RabbitMQ’s documentation 6](#_Toc160269181)

[Q4: How can a message broker be implemented in the context of the project? 7](#_Toc160269182)

[Source 1: Domain model 7](#_Toc160269183)

[Source 1: Component test 8](#_Toc160269184)

[Conclusion 10](#_Toc160269185)

[Q4: Are there any pitfalls to using a message broker? 10](#_Toc160269186)

[Source 1: Expert interview 10](#_Toc160269187)

[Source 2: Component test 11](#_Toc160269188)

[Conclusion 12](#_Toc160269189)

[Research conclusion 12](#_Toc160269190)

[Resources 12](#_Toc160269191)

# Problem

I am a semester 6 student and need to build an enterprise grade scalable system. I plan to use microservices for my different application modules. However, they should not interact with HTTP as they need to be decoupled and they need to scale independently.

# Opportunity

This brings forward the opportunity to research popular ways in which microservices interact with each other and whether a message broker solution can cover the use case of this semester. This will be able to clarify an unknown area in my knowledge and drive the individual project forward.

# Main research question:

The main research question that this document will try to answer is the following:

How can microservices interact with each other and still be decoupled?

An answer to the main question is going to be derived from answering all of its associated sub-research questions:

1. What are the most popular ways to decouple microservices?
2. What is a message broker?
3. How does a message broker guarantee that messages are delivered?
4. How can a message broker be implemented in the context of the project?
5. Are there any pitfalls to using a message broker?

# Q1. What are the most popular ways to decouple microservices?

## Source 1: Good and bad practices

As explained by Saurabh Gupta[1], decoupling software architecture is a design approach that aims to enhance flexibility, scalability, and maintainability by reducing dependencies between different modules or services of a software system. It encourages the use of modular design, in which each component is made to work independently and communicate with one another via well specified interfaces.

The most notable aspects of a decoupled system are:

1. Modularity
2. Loose coupling
3. High cohesion
4. Scalability
5. Sped up development and testing

There is a common saying that if one person cannot build a stable monolith, they should not even try to approach microservices. This is due to the fact that with a microservice architecture, each component need to have a very clear logical separation so that it can be loosely coupled to other modules. If there is no easy way to separate it, it’s probably not meant to be separated, which proves that one must first be able to build a well-structured monolith, which can then be split up into decoupled modules.

A common approach[1] (outside of the inherent logical separation between the modules) is using a message broker to facilitate the communication between microservices. Message brokers use queues, which are consumed asynchronously and do not lead to direct HTTP failures. A diagram of a application

Description automatically generated

Figure 1 - popular way of decoupling microservices

A message broker, along with logically split modules leads to the loosely coupled nature of a microservice architecture that is sought after.

## Source 2: Practical work examples

During last semester’s internship, even though I did not get to work on a microservice application specifically, the company had a big enough project where it was worth it to invest into splitting up the application’s modules.

When the company mentor was asked about it, he Kees van der Broek, owner of D-centralize B.V. said the following:

“If an application is not big enough, it’s not even worth it to consider microservices. They are quite costly and can become tricky to work with.”

A person standing in front of a whiteboard

Description automatically generated

Figure 2 - Interview session about microservices with Kees van der Broek, owner of D-centralize B.V.

However, in the interview (Fig. 2) he also noted that:

“However, at a certain point when the application grows you start to notice pieces of the project that you can break off into their own things. You can afterwards scale these pieces for the application, benefit.”

Exploring some of the documents of the project’s architecture, it became clear that they also take advantage of a message broker (in their case Gcloud Pub/Sub) to promote asynchronous event-driven interaction between the project’s microservices.

## Conclusion

The most popular ways to de-couple microservices lie in their ground concept. In order to separate a part of the application into a module, it needs to hold a clear logical piece to the puzzle that can be scaled on its own. If such a piece cannot be defined, then it does not exist and microservices should not even be considered.

As for the facilitation of loose coupling between the modules, message brokers play the role in achieving event-driven asynchronous interactions between an application’s microservices.

# Q2: What is a message broker

## Source 1: Library research

As described by IBM[2], “A message broker is software that enables applications, systems, and services to communicate with each other and exchange information. The message broker does this by translating messages between formal messaging protocols.”

This allows different modules to communicate with each other directly, even if they are written in different languages or implemented in different platforms. What’ also an important description is the following:

“Message brokers can validate, store, route, and deliver messages to the appropriate destinations. They serve as intermediaries between other applications, allowing senders to issue messages without knowing where the receivers are, whether or not they are active, or how many of them there are. This facilitates decoupling of processes and services within systems.”

## Source 2: exploring RabbitMQ’s documentation

RabbitMQ is used widely by enterprise applications[4]. It has the ability to store messages in queues, can be deployed with Kubernetes and is enterprise and cloud ready.

According to RabbitMQ’s documentation[3], they describe their product with a few lines:

“Asynchronous messaging – including multiple protocols, message queuing, acknowledgement and flexibility”

it enables software applications to connect and scale. Applications can connect to each other, as modules of a larger application, Messaging is asynchronous, decoupling applications by separating sending and receiving data. It can be used for data delivery, non-blocking operations or push notifications, or publish / subscribe, asynchronous processing and work queues. All these are patterns form part of messaging.

## Concluson:

A message broker is a piece of software that enables the communication between two different systems (or system modules) even if they are written in different languages and utilize different technologies. In RabbitMQ’s case, this messaging is asynchronous and event-driven, allowing effective decoupling between application components.

# Q3: How does a message broker guarantee that messages are delivered?

## Source 1: Expert interview

Coming back to the interview (Fig. 2) with Kees van der Broek, he notes the following:

“There is no guarantee that messages will always be delivered. That’s the tricky thing about it. You have to make sure that your microservice does not entirely hinge on receiving messages”

What is guaranteed, however is that messages are sent and are stored in queues, ready to be delivered to anyone that is subscribed. If the message broker supports message acknowledgment, these messages reside permanently in the message queue until they can be resolved.

Some message brokers even implement message persistence, ensuring that even in the case of a server shutdown, they are saved on the machine’s hard drive, which makes them persist across machine malfunctions.

## Source 2: RabbitMQ’s documentation

RabbitMQ is a very flexible and popular messaging solution[4], that has mechanisms implemented to ensure messages are sent and delivered. As their documentation[3] notes:

“RabbitMQ offers a variety of features to let you trade off performance with reliability, including persistence, delivery acknowledgements, publisher confirms, and high availability.”

Just like Kees mentioned in the previous interview, RabbitMQ specifically supports message persistence on the machine’s hard drive, as well as durable message queues and message acknowledgment.

## Conclusion:

A message broker guarantees message delivery through internal mechanisms implemented by the maintenance team. In the case of RabbitMQ[3] it is achieved through durable queues and exchanges, as well as message acknowledgment and persistence on the machine’s hard drive.

# Q4: How can a message broker be implemented in the context of the project?

## Source 1: Domain model

In order to find out how a message broker can be implemented in this semester’s project, the domain of the application was modelled in a C2 diagram (Fig. 3) to get a better understanding of how the different components would interact with each other.

A screenshot of a diagram

Description automatically generated

Figure 3 - Initial C2 draft of HouseHunters' architecture

As it can be seen in the C2 diagram (Fig. 3), a message broker can be used for event-driven communication between the different microservices of the application. Each microservice has its own database and manages its own entities.

Whenever an update to these entities is made, interested listeners who are subscribed to the respective queue can act accordingly once they receive the action that has been performed.

In the case of the listing microservice, it is interested in when media is uploaded to the S3 bucket, so that it can take the object keys and store them in its own database.

The media microservice, on the other hand, is interested in when a listing would be deleted. In that case it would delete its associated media in the bucket.

The bid microservice is interested in any changes that happen to a listing, whenever one is created, edited or deleted, the bid microservice can save a local fragment as a reference to when a bid is being placed.

## Source 1: Component test

In order to test the domain model, a small-scale test can be performed between the listing and bid microservice. This would serve as a walking skeleton and proof that this concept can work on a larger scale.

The setup is as follows:

* A docker container running RabbitMQ
* The listing microservice
* The bid microservice

The listing microservice would declare an exchange “listing\_created” to which a queue will be bound. The queue “listingCreatedBidSub” serves as a subscription where the bid microservice would receive information about the listing that was created.

The first step of the test is to have the exchanges and queues declared as follows:

async function connectToRabbitMQ() {

    try {

        const connection = await amqp.connect('amqp://localhost');

        channel = await connection.createChannel();

        // Create exchange in case it does not exist

        await channel.assertExchange('listing\_created', 'fanout', {

            durable: false

        });

        await channel.assertQueue("listingCreatedBidSub", { durable: false });

        await channel.bindQueue('listingCreatedBidSub', 'listing\_created', '');

        console.log("Connected to RabbitMQ");

    } catch (error) {

        console.error("Error connecting to RabbitMQ:", error);

    }

}

This would allow the bid microservice to listed for events published to the listing\_created exchange. The exchange is of type fanout, which means that all queues bound to it will receive the message.

Afterwards, in the listing microservice, once a listing is successfully saved in the database, its details can be published as a message in the exchange:

    const listingInfo: createListingPayload = req.body;

    const listing = await postListing(req.userId, listingInfo);

    const listingDto = mapListingToDTO(listing);

    channel?.publish('listing\_created', '',Buffer.from(JSON.stringify(listing)));

    const response = {

      message: 'Listing created',

      listing: listingDto

    }

    res.status(200).json(response);

Following that, the function that listens to the event on behalf of the bid microservice looks as follows:

export async function subToExchanges(channel: amqp.Channel) {

  channel.on("error", (error) => {

    console.log("Error occurred during message consumption:", error);

  });

  await subToQueue(channel, "listingCreatedBidSub", handleListingCreated);

}

async function handleListingCreated(channel: amqp.Channel, msg: amqp.Message) {

  const eventData: IListingFragment = JSON.parse(msg.content.toString());

  console.log("RECEIVING LISTING CREATION EVENT: " +  eventData)

  (channel as amqp.Channel).ack(msg);

}

Once the applications are started and the bid microservice is listening, a request in postman can be sent containing the information to create a listing. Once the listing is successfully created, the following can be seen in the console (Fig. 4):

A screen shot of a computer

Description automatically generated

Figure 4 - Output terminal of a microservice receiving an event

## Conclusion

A message broker can be implemented in the context of the project though a “fanout” type exchange and messaging queues. As described in the C2 diagram (Fig. 3), it can be used to allow the different microservices to communicate with each other in an event-driven way, thus separating them while still retaining a loosely coupled nature.

Should one of the microservices go down, the others will be unaffected and messages will be piled up in the queue, awaiting for the subscriber’s availability.

# Q4: Are there any pitfalls to using a message broker?

## Source 1: Expert interview

Once again referring to the interview with Kees van der Broek (Fig. 2), he noted:

“The main trouble with using messaging brokers is that once communications start to pile up, it’s hard to track what is going on. One event can spawn 10 others, and with their asynchronous nature you start to lose track”.

The big pitfall with message brokers and microservices seems to be complexity. Once the system becomes large, there can be dozens of microservices and hundreds of exchange queues. It would be really difficult to monitor such a system, since communication is not streamlined anymore like HTTP.

It is important to implement robust system monitoring and error handling to keep track of things that go wrong.

## Source 2: Component test

Going back to the walking skeleton example, a way to test this complexity would be to tie together a bunch of message interactions that spawn more down the line, resulting in a tree-like structure of events.

In the code where the bid microservice receives the information about the created listing, it is going to publish a message on its own further broadcasting the created listing.

  const eventData: IListingFragment = JSON.parse(msg.content.toString());

  channel.publish("listing\_created\_received", '', eventData)

  (channel as amqp.Channel).ack(msg);

Afterwards, the authentication and media microservice are going to listen for the event. The authentication microservice will play an important role, because it will try to parse the ObjectId of the listing.

async function handleListingCreated(channel: amqp.Channel, msg: amqp.Message) {

  const eventData: IListingFragment = JSON.parse(msg.content.toString());

  const objectId = new ObjectId(eventData.listing.\_id)

  await saveListingFragment(eventData);

  (channel as amqp.Channel).ack(msg);

}

To test what is going to happen now, the listing microservice will purposefully publish a message containing a listing with an invalid ObjectId. What happens is the following error in the terminal of the authentication microservice (Fig. 5):

A computer screen with text

Description automatically generated

Figure 5 - Untraceable error being thrown in the authentication microservice's termianl

Where did this error come from? Did it happen internally? We are not even informed that it happened from a result of a received message. In this case, it would be really hard to track that a message with a wrong ObjectId has been passed down between several microservices before creating an error down the line.

## Conclusion

The main pitfall of implementing a message broker solution is complexity. When more microservices and exchanges of messages start to appear, it becomes really difficult to track if something goes wrong with the system. For this reason, it is important to implement robust system monitoring and error handling.

# Research conclusion

Through gathering all of the results and from the answered sub-questions, a conclusion can be drawn and the main question can be answered.

**How can microservices interact with each other and still be decoupled?**

For the project HouseHunters in semester 6 advanced software, microservices can interact with each other and still be decoupled through several means. The main approach is to follow best established practices in the industry for architectural decoupling of individual semester components, while still achieving loose coupling through using a message broker.

The most important one is to have a logical separation between the different application modules. After all, if a specific microservice does not have a clearly defined purpose and functionality of its own, it should not even exist in the first place. This separation can be defined even more clearly by having each microservice operate on its own data storage.

Once clear logical separation has been achieved between the microservices, RabbitMQ can be used to facilitate event-driven asynchronous communication, enabling loose coupling of the different microservices. Exchanges of type fanout can be used for publisher microservices to communicate events that can be picked up by any microservices that is subscribed, where they can save any relevant data they are interested in on their own implemented data storage mechanism. All of the exchanges and queues need to be durable and persistent to avoid the loss of messages.

# Resources

1. Gupta, S. (2023, September 9). Decoupled Architecture & Microservices - Saurabh Gupta - Medium. *Medium*. https://medium.com/@saurabh.engg.it/decoupled-architecture-microservices-29f7b201bd87
2. *What are Message Brokers? | IBM*. (n.d.). https://www.ibm.com/topics/message-brokers#:~:text=the%20next%20step-,What%20is%20a%20message%20broker%3F,messages%20between%20formal%20messaging%20protocols
3. *RabbitMQ: easy to use, flexible messaging and streaming | RabbitMQ*. (n.d.). <https://rabbitmq-website.pages.dev/>
4. Jaylin. (n.d.). *RabbitMQ vs Kafka: 5 Key Differences & Leading Use Cases*. www.emqx.com. https://www.emqx.com/en/blog/rabbitmq-vs-kafka