

# *Integrating multiple export formats with microservices using a service bus.*

Jesse Wissink

Saxion University Enschede

A microservice solution for process automation.

## Information

*Integrating multiple export formats with microservices on a service bus.*

A microservice solution for process automation at Semso software development in Lichtenvoorde, the Netherlands.

### Student

Jesse Wissink

Student at Saxion University of Applied Science.

Student number: 315769

e-mail: [jwissink@gmail.com](mailto:jwissink@gmail.com)

### Supervisor

Paul Bonsma

Teacher / researcher at Saxion University of Applied Science.

e-mail: [p.s.bonsma@saxion.nl](mailto:p.s.bonsma@saxion.nl)

### Company supervisor

Niek Rassing and Lucas Heezen

Owners and project leaders at Semso software development

e-mail: [niek@semso.nl](mailto:niek@semso.nl)

[Lucas@semso.nl](mailto:Lucas@semso.nl)

## *Summary*

Semso produces custom made software solutions catered to the needs of the clients who need an integration between suites. These suites often export database data, which Semso in turn needs to convert to a required format, for example, PDF. After making these solutions, Semso felt that a general solution for this process is needed to avoid repetitive tasks.

Currently Semso utilizes a custom designed and implemented service oriented architecture (SOA). To make a streamlined workflow and a generic platform on which to deploy the operations Semso requested a design for a microservice oriented architecture with a message broker in between, commonly referred to as an enterprise service bus (ESB).

As assignment for both the educational institution and Semso is to design a well-structured microservice architecture in accordance to common practices.

One of the frequent tasks required by the clients of Semso is for instance generating a PDF document based on an XML file. Semso currently has a service running that automates this process, yet it is not optimized for scaling.

In this document you can find a research approach to a microservice architecture in which existing libraries and recommendations are used to achieve the goals defined further in the document. Examples of these objectives are continuous deployment and hot-plugging.

## *Preface*

When you know the essence of programming code, I think the next software related assignment will be another adaptation of your skills, regardless of the product.

With this being mentioned, as a graduation internship I specifically chose to go out of my comfort zone and delve into other areas covered by programming. Especially the areas of software development and design, embedded software and gaming. This process has resulted in a challenging assignment at Semso.

To realize the design, I would like to thank Leon de Rooij for his explanation of multiple related topics and his experience with certain tools as well as the practical insight into continuous development, deployment and integration as well as secure socket layers.

My thanks also go out to the teachers at Saxion University of Applied Science and Semso for giving me the building blocks to prepare for this assignment and supporting me in broadening my awareness in multiple areas related to software development.

# Contents

<b>INFORMATION .....</b>	<b>2</b>
<b>SUMMARY.....</b>	<b>3</b>
<b>PREFACE .....</b>	<b>4</b>
<b>1. INTRODUCTION AND DEFINITION.....</b>	<b>7</b>
ENTERPRISE SERVICE BUS.....	8
COMMON CHARACTERISTICS OF MICROSERVICES .....	9
<b>2. TERMS OF SATISFACTION .....</b>	<b>12</b>
MODULAR INTEGRATION.....	12
MONITORABLE DATA FLOW .....	12
EXISTING LIBRARY USE .....	12
UNIT TESTING .....	12
INTEGRATION TESTING.....	13
CONTINUOUS DEPLOYMENT.....	13
EXTERNALLY MONITORABLE .....	13
MICROSERVICE CONFIGURATION.....	13
HOT-PLUGGABLE .....	14
<b>3. PROBLEM .....</b>	<b>14</b>
SUB-QUESTIONS .....	14
<b>4. SCOPE.....</b>	<b>15</b>
INCLUDED .....	15
EXCLUDED.....	15
CONSIDERED.....	15
<b>5. PRELIMINARY DESIGN .....</b>	<b>16</b>
<b>6. RESEARCH RESULTS .....</b>	<b>17</b>
SERVICE BUS INTERACTION.....	17
<i>RabbitMQ</i> .....	18
<i>AMQP</i> .....	19
MONITORING .....	19
<i>Nagios</i> .....	19
<i>Paessler Router Traffic Grapher (PRTG)</i> .....	19
<i>Dynatrace</i> .....	19
<i>Grafana</i> .....	20
LOGGING .....	20
<i>Logstash</i> .....	20
<i>Splunk</i> .....	20
<i>Logalyze</i> .....	20
<i>Graylog2</i> .....	20
<i>Elastic Search</i> .....	20
SCALABILITY AND LOAD BALANCING .....	21
<i>NGINX</i> .....	21
<b>7. CONCLUSION AND DISCUSSION .....</b>	<b>22</b>
CONTINUOUS DEPLOYMENT .....	22
MESSAGE BROKERING.....	23
LOAD BALANCING .....	23
HOT-PLUGGABLE .....	24

MONITORING ..... 24

LOGGING ..... 25

UNIT TESTING ..... 25

PLATFORM ..... 25

MEASUREMENT OF SUCCESS ..... 27

**8. FURTHER STUDIES ..... 27**

**9. PRODUCTS..... 28**

LOAD BALANCING, HOT PLUGGING AND MODULARITY..... 28

CONTINUOUS DELIVERY ..... 30

MICROSERVICE CONFIGURATION..... 31

**10. APPENDIX I – BIBLIOGRAPHY ..... 31**

## 1. Introduction and definition

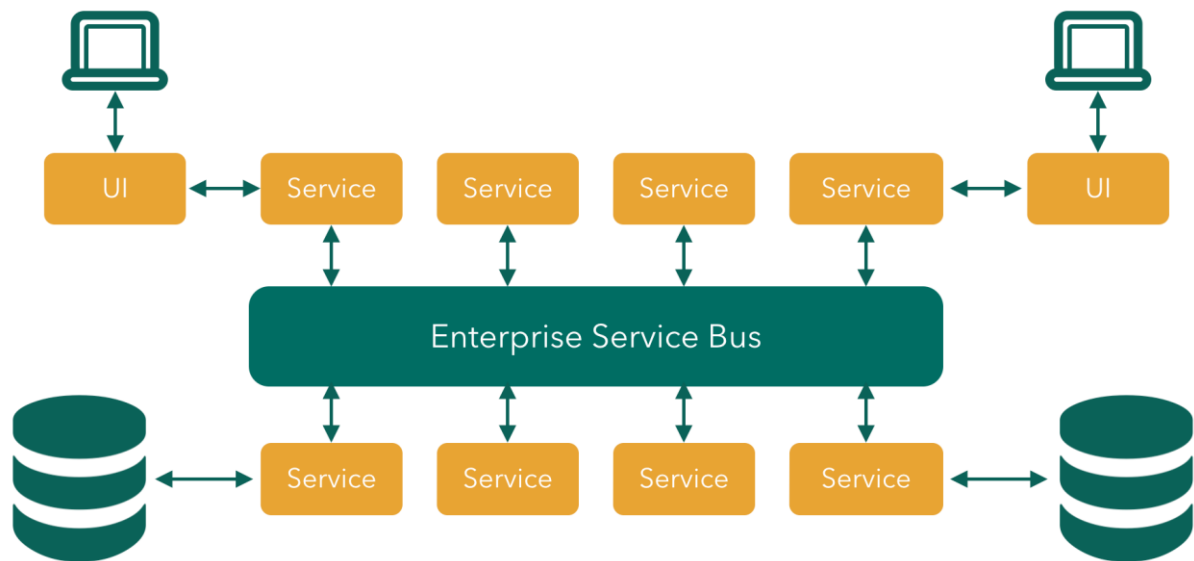


Figure 1. An example of a microservice architecture

When designing software from the ground up, regardless of the application, a platform has to be designed and developed. The platform can be designed in accordance with many design patterns. A popular and new approach to design new software is a microservice approach.

In contradiction to the standard well known single big application, also referred to as a “monolithic” application, a microservice oriented architecture splits up all the operations performed in small services that transfers the data to other microservices by means of either direct HTTP service to service communication as seen in Figure 2 or with an enterprise service bus between as message broker using the advanced message queue protocol as seen in Figure 3.

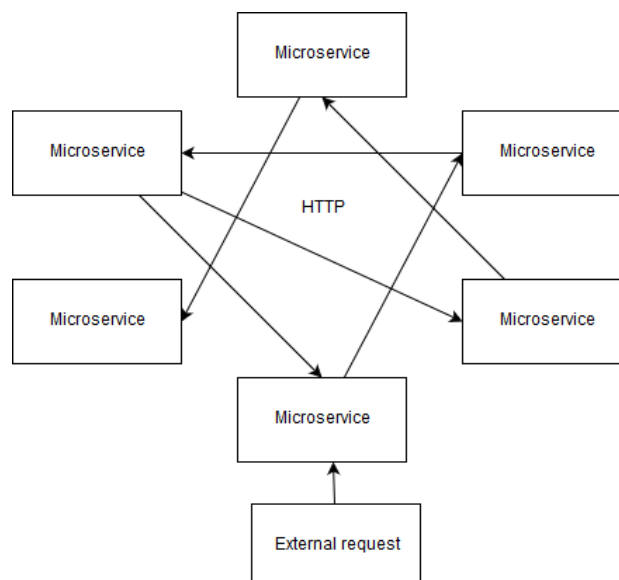


Figure 2. Microservice architecture without ESB

Since a multitude of features can be designed in a microservice architecture, the scope has been reduced with a set of demands. These demands will be further explained at chapter 2.

“Microservices are an approach to distributed systems that promote the use of finely grained services with their own lifecycles, which collaborate together. Because microservices are primarily modeled around business domains, they avoid the problems of traditional tiered architectures (monolithic applications). Microservices also integrate new technologies and techniques that have emerged over the last decade, which helps them avoid the pitfalls of many service-oriented architecture implementations.” (Newman, 2016)

### *Enterprise service bus*

An Enterprise service bus (ESB) in a microservice environment acts as the main pipeline of communication between services, often referred to as a message broker or message queue and described as “an integration and orchestration of microservices” (Wähner, 2016). A visual example can be seen on Figure 3.

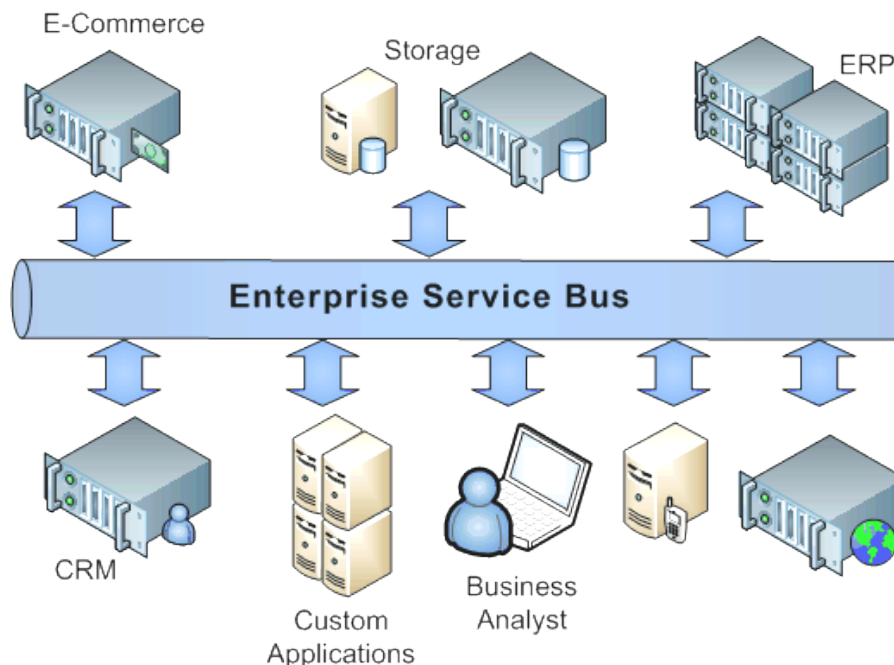


Figure 3. An example of an Enterprise Service Bus (ESB)

Customary for a service bus is to obtain all messages that have been posted on the service bus and relay these to the subscribers of this event.

According to IBM (2012) an ESB is required to handle the following topics:

- **Routing**  
Routing messages between services. An ESB offers a common communication infrastructure that can be used to connect services, and thereby the business functions they represent, without the need for programmers to write and maintain complex connectivity logic.
- **Converting**  
Converting transport protocols between requester and service. An enterprise service bus provides a consistent, standard-based way to integrate business functions that use



different IT standards. This enables integration of business function that could not normally communicate, such as to connect applications in departmental silos or to enable applications in different companies to participate in service interactions.

- **Transforming**  
Transforming message formats between requester and service. An ESB enables business functions to exchange information in different formats, with the bus ensuring that the information delivered to a business function is in the format required by that application.
- **Handling**  
Handling business events from disparate sources. An ESB supports event-based interactions in addition to the message exchanges to handle service requests.

Aside from the required operations, an ESB also provides the following features:

- **Location and identity**  
Participants do not have to know the location or identity of other participants. For example, requesters do not have to be aware that a request could be serviced by any of several providers; service providers can be added or removed without disruption
- **Interaction protocol**  
Participants do not have to share the same communication protocol or interaction style. For example, a request expressed as a simple object access protocol (SOAP) over HTTP (Hypertext Transfer Protocol) can be serviced by a provider that understand SOAP over Java Message Service (JMS).
- **Interface**  
Requesters and providers do not have to agree on a common interface. An ESB reconciles differences by transforming requests and response messages into a form expected by the provider.
- **Qualities of interaction service**  
Participants, or systems administrators, declare their quality-of-service requirements, including authorization of requests, encryption and decryption of message contents, automatic auditing of service interactions, and how their request should be routed.

### *Common characteristics of microservices*

According to Fowler (2014), the following points are common characteristics of microservices.

1. **Componentization via services**  
Given the close resemblance to a component driven architecture (a software design where operations are segmented by components), it also shares the characteristic of being independently replaceable and upgradeable. In practice this means that a microservice can be upgraded and/or replaced without interrupting the overall process and can be deployed autonomously.
2. **Organized around business capabilities**  
The microservices should represent departments and/or operations done in the real world. For example, in an order and shipping company one can expect a microservice for ordering, shipping and payment.
3. **Products not projects**  
Each microservice should deliver a 'product' instead of a project part. This closely ties in with the independence of a microservice.
4. **Smart endpoints and dumb pipes**  
The protocol of communication over the bus should be relatively simple, and the complexity of the operations in the microservices themselves.
5. **Decentralized governance**

Instead of having a process that governs the flow of data and error handling, in a microservice architecture, the microservices are governing themselves.

6. Decentralized data management  
Like the governance, the microservices are responsible for their own data. For example, every microservice could have their own database.
7. Responsible for own persistence  
Since the governance is decentralized, it is recommended that the microservices themselves are responsible for their persistence. In practice, this could translate to an error message on the bus when it crashes or goes offline, which can cause it to restart.
8. Never interact with a different microservice's databases, only through API.  
Messages on the bus should never be a query to another microservice's database.
9. Microservice language chosen based on focus.  
The programming language used in the microservice should be based on the microservice's operations and which language is most efficient to deal with those operations.
10. Infrastructure automation.  
Given the microservice autonomous management and persistence, an environment is created which keeps itself up and running.
11. Design for failure.  
A common design practice for microservices according to Fowler (2014), one should develop their microservices with the assurance in mind that they can, and will break.

Both a monolithic (one single big application) and a microservice approach gives certain advantages. A short list of these can be seen in Table 1.

Monolithic architecture	Microservice architecture
Simplicity.	Partial Deployment
Consistency.	Availability
Inter-module refactoring.	Preserve Modularity
Instant data transfer.	Multiple Platforms
	Deployment speed
	Rapid refactoring.
	Network protocol technology.

Table 1. Advantages of a monolithic and microservice architecture

As a guideline, the following conventions are recommended Bogard (2016) and Netflix (2016) promote a consistent communication protocol between services.

- Caching database operations
- Visualization of the process.

A microservice environment, according to (Bogard, 2016) and (Netflix, 2016), mirrors, or should mirror the existing business structure. If the business structure does not reflect the microservice environment, usually either the company structure follows, or the microservice environment.

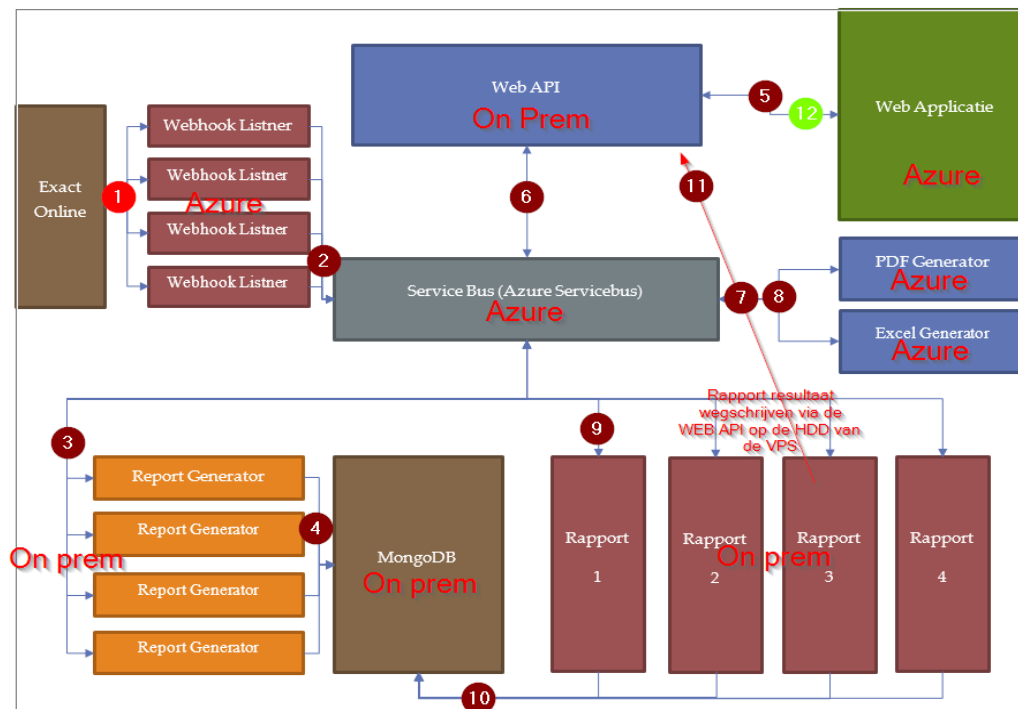


Figure 4. Current microservice architecture at Semso

A diagram of the current running microservice environment at Semso can be seen on Figure 4. In this diagram the incoming data is from a known database provider named Exact Online. Semso receives the data via means of web hook listeners. These fires when changes are made to databases it is connecting to. The incoming data is passed to the service bus which then delivers it to the report generators. These report generators transform the data to an inhouse format and saves it to a Mongo database.

From the web application a client can request a document formatted to selected data types. When a request is made, the web application reports this to the Web API, which redirects this over the service bus to the report generators. These generators save a report, prepared for the export format, on the hard disk and sends the location back to the web application. The web application displays the report as final pass. The numbers in Figure 4 represents the sequence of actions.

## *2. Terms of satisfaction*

To achieve a complete design within the given time of 20 weeks, the following terms of satisfaction have been defined in two sets. “insufficient” and “sufficient”.

One of the objectives of the architecture should be to import data in a predetermined format and export it to a required format, for example, XML or PDF and should have the following functionality:

### *Modular Integration*

Modular integration in this context means that multiple clients can be implemented in the platform at any time, which closely ties in with hot-pluggable. Modular integration can also be referred to as being multi-tenant. To prove modular integration, a test case will be written that will add extra functionality to the platform

#### **Insufficient**

Upon implementing a microservice that transforms the data from the source to a different microservice, the architecture either halts, doesn't respond.

#### **Sufficient**

Upon implementing a microservice that transforms the data from the source to a different microservice, the existing microservices do not change behavior.

### *Monitorable data flow*

To prove that the data flow is monitorable, a test case will be written that, by means of graphics or text, displays the flow of information. To prove that it is an understandable scenario, an employee at Semso will review the monitoring software, and explain what is shown without having prior knowledge of the monitoring software.

#### **Insufficient**

The reviewer did not understand what was shown or written, or there is no output at all.

#### **Sufficient**

The reviewer either instantly recognized the data flow or understood the data flow after minimal instructions.

### *Existing library use*

Under the notion of not reinventing the wheel, research will be done in libraries that can be used in the test cases. This will be graded based on how many of the terms can be realized with external libraries

#### **Insufficient**

0 – 4 test cases use external libraries.

#### **Sufficient**

5 – all test cases use external libraries.

### *Unit testing*

The workflow to delivery should also include unit testing.

**Insufficient**

There is no way to unit test written software or the tool has to be developed.

**Sufficient**

A tool or IDE will offer means of unit testing.

### *Integration testing*

To prove that all microservices interact with the architecture according to the design, an integration test will be written. This test will report on the interaction of the microservice with the service bus, the service bus load and message retention. All deployed microservices should react and report to the integration test.

**Insufficient**

When the required microservices are attached to the service bus, there is either a lack of data or communication with the service bus. Also, the report generated is either insufficient, inaccurate or non-existent.

**Sufficient**

All the deployed microservices respond to the written integration tests and report accurately on the communication with the service bus, incoming and outgoing. From this, a well-structured report is generated.

### *Continuous deployment*

To prove that new microservices can be deployed without interrupting the system and/or architecture, a test scenario is written where new microservices are introduced to the platform.

**Insufficient**

Upon deploying a new microservices in the platform, either the service bus or the microservice errors on service bus interactions or it interrupts the system.

**Sufficient**

Without any interrupts in the current service bus or existing microservice operations, the new microservices integrates flawlessly.

### *Externally monitorable*

To prove that the architecture can be monitored off-site, a test scenario that will send the monitor data to an external receiver. To conform to the microservice methodology, it will be the same monitoring microservice / interface used to test it internally.

**Insufficient**

Either the architecture cannot be monitored off-site or is inaccurate. This test will also prove insufficient if the same microservice as internal monitoring can't be used.

**Sufficient**

Monitoring data is accessible from an external source.

### *Microservice configuration*

To prove that the microservices are configurable without reprogramming the microservice itself a test case will be written where a microservice has to be configured.

**Insufficient**

For any change in the incoming or outgoing formats the microservice has to be reprogrammed. The microservice is not externally configurable and most of the parsing settings are hardcoded.

**Sufficient**

The parsing or transforming behavior can be altered by editing an external file or by a configuration microservice. This file or service will hold information concerning the microservice operations. For example, how to parse the data, which fields to save and with what flag it should be delivered to the service bus and/or from internal to external.

***Hot-pluggable***

To prove the architecture supports hot-plugging, a test case will be written that deploys and removes microservices which handles similar operations. The addition and removal of these microservices should not interrupt the overall process.

**Insufficient**

When new microservices are introduced the overall process is interrupted or does not function.

**Sufficient**

Upon deployment or removal of microservices, the architecture responds by either incorporating or neglecting these in the process without any noticeable effect save for the requested operations.

### ***3. Problem***

The graduation assignment can be described as designing a well-founded microservice architecture. The architecture will be designed according to the recommendations given by leading figures in this field the answers to the question and sub questions can be found at page 22.

“How can a monitored microservice architecture be designed to facilitate data transfer between importing from an external source and exporting in formats on predefined demands in accordance with common practices”?

***sub-questions***

The following questions help to formulate an answer to the main question:

- What are the common practices when designing a microservices oriented architecture?
- Which tools can be used?

## 4. Scope

The following list will show which topics will be included in the research, and which ones excluded:

### *Included*

**Micro-service management**

How the micro services behave and interact with the service bus.

**Service-bus architecture**

How the microservices and service bus interact with one another.

**Architectural layout**

How the architecture will look.

**Monitoring**

Monitoring the data flow in the architecture.

**Microservice modularity**

Setting up identical micro services which do the same operations with different data sources.

**External libraries**

Which libraries are available, needed and what they contribute.

**Unit and integration tests**

Tests that confirm a successful or unsuccessful integration into the architecture.

**Continuous deployment**

Altering the amount of microservices without interrupting the overall process

### *Excluded*

**Alternatives for micro-services**

Which approach might also be adequate for this problem?

**Alternatives for a service-bus**

Are there other design patterns that can be used in coherence with a service bus?

**Versioning**

For handling microservices in development, of different versions. A multitude of libraries and software suits are available, as well as common guidelines.

### *Considered*

**Resilience testing**

Randomly kill running processes and see how the architecture reacts.

**Autonomous life cycles**

When a microservice dies, how can they be relaunched or be aware of their death?

**Machine learning for interpretation of the data**

If something changes in the format of the incoming data, is there a way to program the parser to adapt automatically?

### Environment scalability

When the workload on the service bus and/or microservices becomes too much, new virtual or physical machines can be deployed to share the load.

## 5. Preliminary design

Upon researching microservices in coherence with a service bus, the preliminary design can be seen on Figure 5.

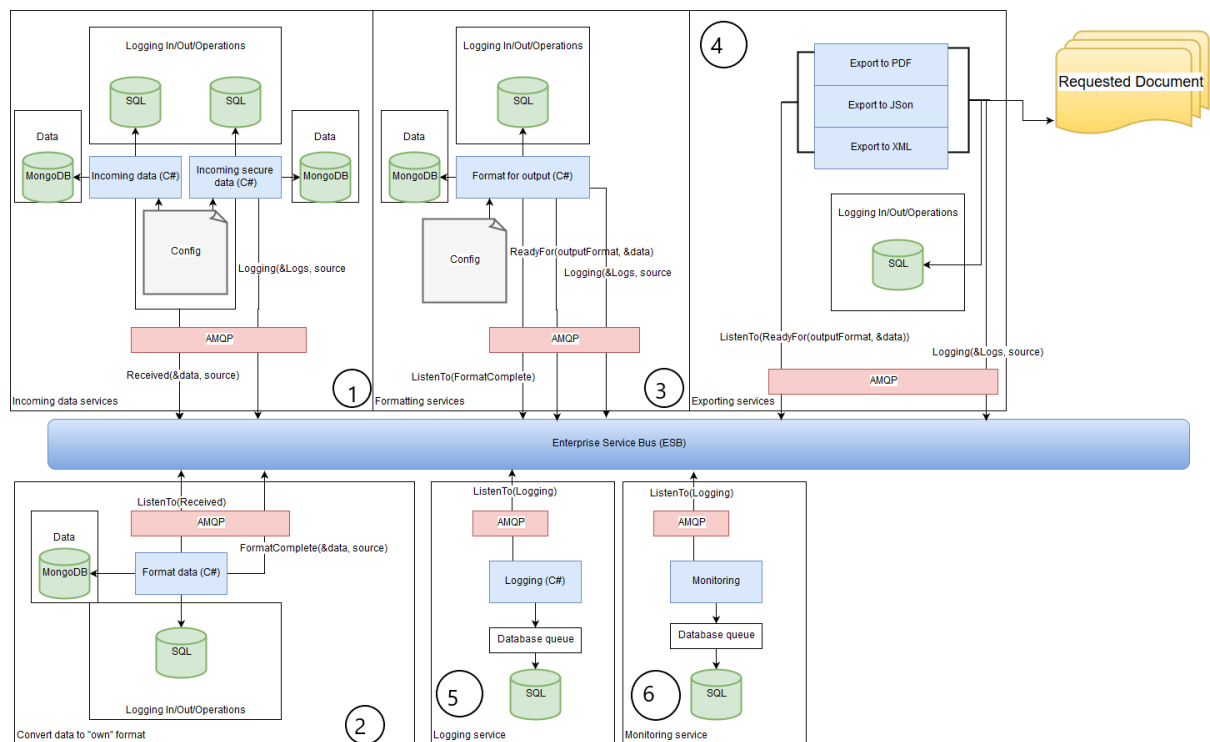


Figure 5. Preliminary design sketch

As seen on Figure 5, the operations required from import to export have been segmented into microservice domains.

1. Incoming data services  
This domain holds the web hooks and callbacks required to import the format from an external source.
2. Converting services  
Converts the data to an architectural friendly format for internal microservice communication.
3. Formatting services  
Formats the converted data to a standardized format for exporting formats
4. Exporting services  
Generates a requested output format from the formatted data
5. Logging services



Logs all service bus interactions from the microservices, for example, *“completed data set X and published these on the bus with message Y”*.

#### 6. Monitoring services

Holds the information required to visualize the flow of information in the service bus and listens to the logging data to visualize microservice activity

In accordance with the general guidelines of microservices, every microservice has its own database and is only responsible for accepting the data and publishing it again after the requested operations are complete.

## 6. Research results

*“There are no passengers on spaceship earth. We are all crew”* -Marshall McLuhan

The operations needed to export a document in a required format can be seen on Figure 6. Where the data is constantly being extracted from an external source, then formatted for MongoDB.

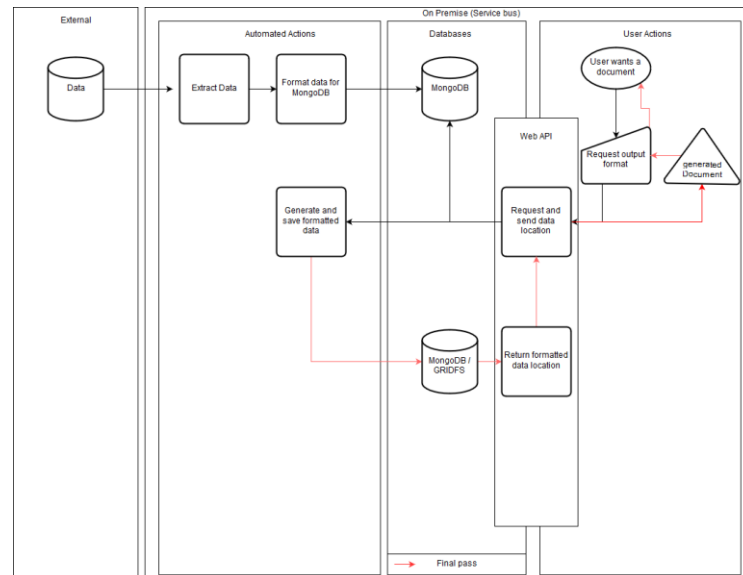


Figure 6. Microservice operations by Semso

On the other end, the user can request a document based on this data, the web API gathers the required information and generates a new format confirmed with the user requested topics.

This is saved into a different Mongo database, and from there a document is generated and the location is send to the requester.

To achieve a well-structured system with extended functionality and complexity, it is recommended to discover what has already been developed. In this chapter, software will be covered which conforms to the scope of this research. The amount of software available for logging, monitoring and scaling is extensive. The tools which are analyzed are recommended by leading figures, personal friends and/or the frequency of appearance.

### Service bus interaction

On recommendation, concerning service bus interaction, Docker was mentioned in combination with RabbitMQ. Initially, Docker is supplying the runtime environment and RabbitMQ provides the service bus with AMQP as protocol.

For a short list of service bus providers, one can navigate to (Wähner, Choosing the right ESB for your integration needs, 2013). As mentioned, further details on ESB alternatives is outside of the scope of this document.

### *RabbitMQ*

The enterprise service bus, also described as a message broker, RabbitMQ receives messages from a publisher and forward these to a message consumer via means of a queue. There should always be a message consumer, if this is not the case, the message broker, in this case RabbitMQ, stores the messages in the queue. When the message broker receives a message, it sends a confirmation back to the sender.

The message consumers give an acknowledgement back to the message broker (queue) when a message is accepted. When there are multiple message consumers for the same type of messages, they will be handled in parallel. An example handling 6 messages over 3 consumers can be seen in Table 2.

Consumer	Message
Consumer 1.	1 and 4
Consumer 2.	2 and 5
Consumer 3.	3 and 6

**Table 2. Parallel message handling.**

Extended on message broking, with RabbitMQ, the message will not be delivered directly to the queue, instead there is an exchange. This exchange binds to a message queue via a protocol named AMQP (advanced message queuing protocol) (Pivotal, 2017).

## *AMQP*

Instead of direct messaging from a publisher to the message broker, AMQP acts as the middleman. Exposing additional functionality to the message system, for example, frame transfer protocol sizes.

Another mechanism to constrain the transfer, is that every container can define the maximum number of channels that can be used. These channels are independent paths over which the messages can be send, a session then binds these two channels into a multiplex connection. This gives flow control, and thus allows for defined connections to have a higher or lower bandwidth. AMQP also requires TLS (Transport-Level Security) and supports SASL (Simple Authentication and Security Layer) connections, which will secure sensitive data.

The flow control given by utilizing AMQP allows for a scalable environment by means of the message receivers decreasing their “Link credit”. AMQP also easily maps to JSON and vice versa. This is also encouraged due to the JSON usage at Semso.

Furthermore, the data that transfers over the wire is mostly interpreted by a schema descriptor. This schema, in AMQP, is customizable. Which may reduce the size of packages that go over the wire (Vasters, 2015).

## *Monitoring*

For debugging purposes and a better insight in the flow of information and the availability of microservices, a monitoring software can be implemented to visually represent the active, paused and inactive microservices. From the information gathered regarding microservices, Nagios is often the recommended software to do monitoring. Currently at Semso, a monitoring software named PRTG, short for paessler router traffic grapher is running to see which services are running, updating or offline.

## *Nagios*

Nagios, also referred to as Nagios Core is an open source monitoring application that monitors systems, networks and infrastructure. Nagios offers monitoring and alerting services for servers, switches, applications and services.

A downside to Nagios is that it's reactive and not pro-active, meaning, that it only detects and alerts when something has gone wrong, and not when it's about to, yet this can be achieved by setting up tolerances for the norm. When data or operations are acting incoherent or outside of the defined expected norm, an alert can be fired.

## *Paessler Router Traffic Grapher (PRTG)*

Like Nagios, PRTG is a network monitoring tool. PRTG checks the availability and uptime of the microservices and other devices by utilizing the PING sensor. Aside from the microservice status, PRTG also allows for system monitoring.

Due to the current usage of PRTG at Semso and their experience with the software, it already scores higher as a baseline.

## *Dynatrace*

A frequently recurring software name for monitoring microservices, specifically in a container environment, is Dynatrace. Dynatrace discovers the microservice architecture automatically by means of an agent. Dynatrace also removed the burden for developers to build in the template for service discovery or flow logging. Aside from monitoring, Dynatrace also offers CD with automated testing and deployment.

### *Grafana*

Recommended by a personal friend, Grafana is an open source monitoring tool that can visualize, alert and unify data. Supporting a plethora of data sources, both open source and commercial (Grafana Labs, 2017).

### *Logging*

Throughout the research it became apparent that the logging standards should tie in as close as possible with current ruling reporting protocols, like HTTP error reports and ISO 8601 timestamps. According to OWASP (2017), logging assists in identifying problems, monitoring, establishing baselines. These logs are taken from a plethora of sources, like monitoring, system status should be logged, but also all the actions taken by the microservices, service bus and tools.

### *Logstash*

Defined as a tool to collect, process and forward events and log messages. These logs are converted into JSON documents, and stores them in an Elasticsearch cluster. Logstash can be used as a log collection tool to redirect it to monitoring software for example.

### *Splunk*

Advertised as software that can turn machine data into answers by collecting and indexing data, searching and investigating the logs, correlate and analyze, visualize and report, monitor and alert. With external access possibilities.

### *Logalyze*

Log collection, log analysis, compliance reports and alerts are the major features of Logalyze according to their website. The website states that it is optimized for the SOAP web service protocol. Logalyze also offers a user interface to view logs, statistics and reports. (Logalyze, 2017).

### *Graylog2*

Like the other logging software available, Graylog offers collection and processing, analyzation and search, visualization and alerts and triggering. Graylog offers more functionality when purchased. Otherwise, the main platform is open source.

Consistently with the websites of the other logging software developers, it is hard if not impossible to find the feature specifics. This is a risk, since informing the user about the software capabilities and how to use them is key. If the website is unclear, this might be reflected in the software documentation as well.

### *Elastic Search*

Recommended by personal friends and leading figures like Fowler, Elastic search is a document container (database) indexing tool. Optimized for scaling, Elastic Search has been developed with big data in mind.

In this scope, Elastic Search can be used as middleware for visualization, analysis and debugging software.

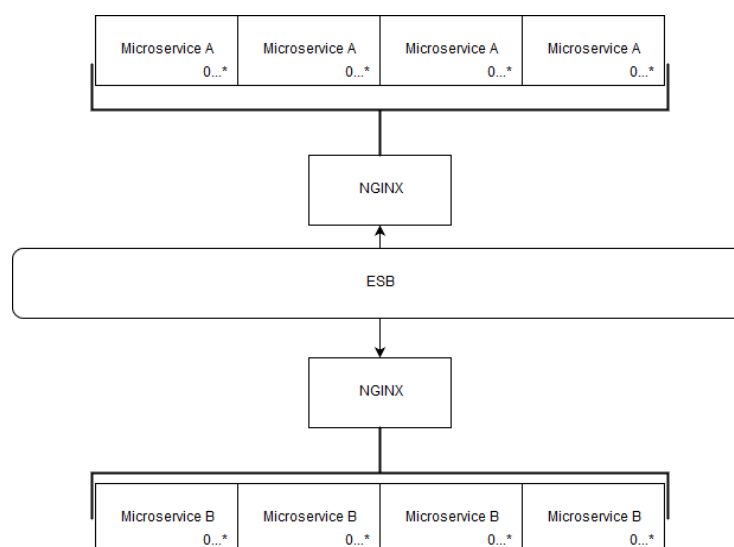
### *Scalability and load balancing*

Due to the expected growth of the company and the unforecastable additions to the architecture. The dynamic scaling of usage should be taken into account. The topic of load balancing and scaling is frequently used in the web development sphere. These load balancers redirect traffic to less busy web services. In the scope of microservices, the load balancing should balance the load over the microservices.

In terms of X-Y-Z scaling (Richardson, The Scale Cube, 2017), scalability on microservices goes mainly in the X-axis (scaling by cloning), taken that microservices are implemented correctly and the Y scaling has been done correctly (scaling down the application by segmenting functionality).

### *NGINX*

Recommended by a personal friend, the leading figures in the field and a frequently encountered topic when researching load balancing and scalability is NGINX (pronounced engine-ex). Most known for running alongside with Apache web server as a reverse proxy. Meaning it can reroute incoming traffic to different IP's. In the scenario of web servers, multiple instances of the same website / web server can be listen under this rerouting table. When applied, NGINX will balance the requests over these listed web- sites/servers.



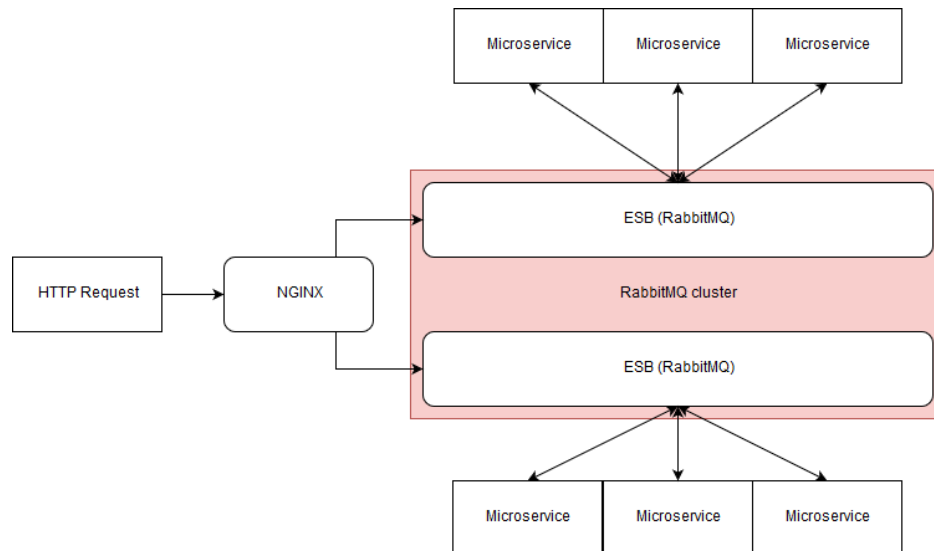
**Figure 7 Interpretation of NGINX**

In Figure 7 an interpretation of how I expected NGINX as load balancing as reverse proxy can be visualized. In this figure it can be seen how the messages from the Enterprise Service Bus pass through the reverse proxy provided by NGINX. This proxy will reroute the traffic through an x amount of microservices. In this way, the load gets balanced over these microservices. The end goal for

implementing NGINX is to dynamically add and destroy microservices based on the load and balance requests accordingly.

A problem, however, is that NGINX uses the HTTP whereas RabbitMQ is utilizing AMQP.

Further research concerning NGINX, RabbitMQ and load balancing, showed that RabbitMQ already balances the messages over all consumers on the same exchange. In the situation of having a RabbitMQ cluster, NGINX can be used to balance the HTTP requests to the message broker. A visualization of that process can be seen on Figure 8, where NGINX balances the incoming HTTP requests over the RabbitMQ cluster, which can consist of multiple ESBs. RabbitMQ



**Figure 8. NGINX balancing over a RabbitMQ cluster**

When using RabbitMQ as a message broker (ESB), the load on the consumers gets balanced by RabbitMQ automatically (Pivotal, 2017). This is done by declaring direct exchanges.

## 7. Conclusion and discussion

Based on the gathered information concerning tooling, practices and principles, a conclusion can be made on multiple topics. The choices for tools are based on the grades received and Semso's experience. The conclusions are listed in order of implementation priority.

### *Continuous deployment*

With the runtime environment being the base of the architecture, it would be recommended to make the runtime environment of the microservice architecture independent of the deployment target. To achieve this, container software like Docker can be used. When a platform supports Docker, it supports the microservice architecture.

Using Docker as environment also greatly reduces complexity that often gets introduced with continuous deployment.

This enables an abstract visualization of the hierarchy which can be seen in Figure 9. The Docker for windows installation completed without errors.

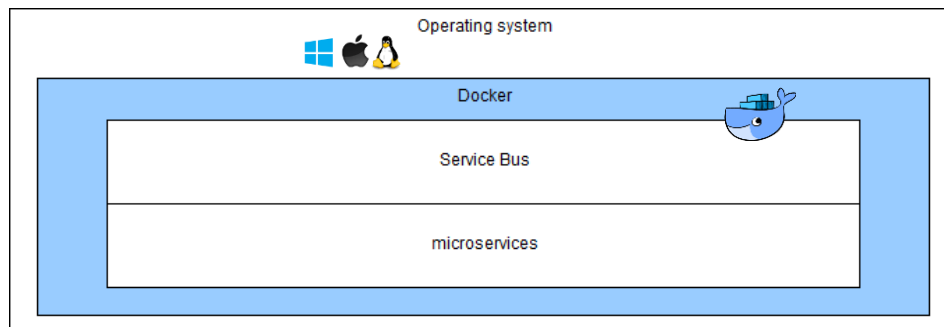


Figure 9. Deployment Hierarchy

Along the research, the difference of using Windows or Linux containers is shown mainly in container availability. The amount of software available for Linux containers widely exceeds the Windows counterparts. For example, RabbitMQ is mainly available for Linux containers, yet to run it on Docker with Windows containers, special versions of the related software is required. In practice this translates to [user-who-made-the-software]\[product-windows-version]:[version-tag]. The problem this creates is amplified in severity when platform complexity increases. For example, when adding logging and monitoring tools for Docker, there has to be a windows container version. It is recommended to run Docker for Windows on Windows with Linux containers to maximize its capabilities.

### Message brokering

With RabbitMQ as ESB, the protocol that will be used to communicate between the microservices is AMQP. The flow of communication from the ESB over the AMQP can be seen in Figure 10.

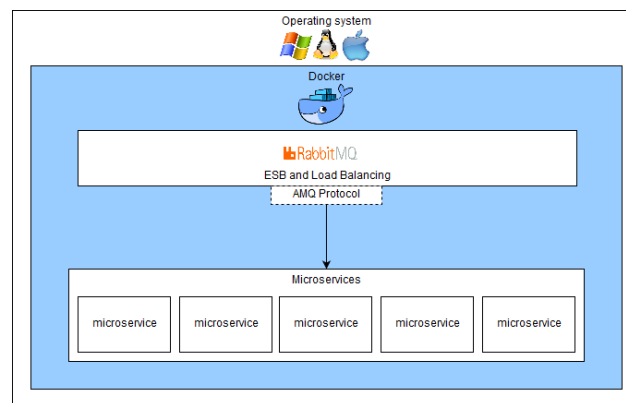


Figure 10. AMQP in hierarchy

With RabbitMQ responsible for load balancing between similar workers, the channel has to be declared with a prefetch count. This can be declared separately to each new consumer on the channel or shared across all consumers on the channel. This enables a worker/consumer to declare when it is ready for a new message.

### Load Balancing

When using Docker as deployment target, a commonly recommended ESB is RabbitMQ. Microservices can publish and receive messages by connecting to the RabbitMQ IP and port, which are assigned by Docker. To ensure consistency in the address, Docker has an internal DNS, which can be used to apply a hostname to running containers. In this scenario RabbitMQ will also act as load

balancer. The visualization of the addition of RabbitMQ can be seen in Figure 11 and the proof of concept (PoC) in chapter Load Balancing on page 28.

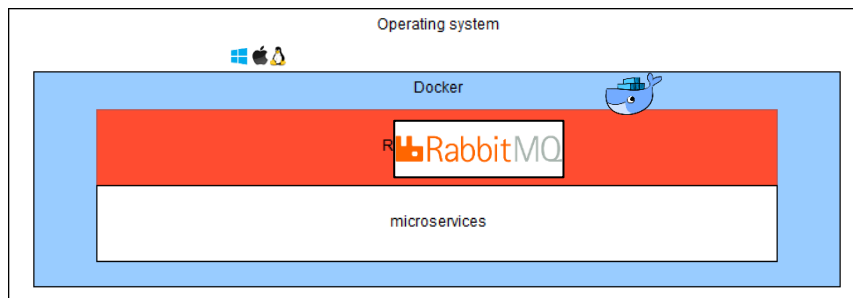


Figure 11. Service Bus in hierarchy

## Hot-pluggable

The PoC's on page 28 also proves hot-plugging and can be added in the following situations:

- **Fanout Exchange**  
With the fanout exchange, where messages will be delivered to all subscribed consumers, hot plugging is a feature that comes by default. In this scenario it is possible to add and remove microservices to the exchange without interrupting the overall process.
- **Direct Exchange**  
With the direct exchange, a routing key can be defined for the queue. Based on the routing key, messages with that key will be delivered to subscribers of that key like a fanout. Hot pluggable is there for enabled due to subscribers being able to listen to pre-defined routing keys or creating a new routing key process with a publisher and subscriber over the same exchange. By default, if only the queue is declared, RabbitMQ will bind these queues over the AMQP default direct exchange.

In situations where a fanout exchange has to be load balanced at one of the endpoints, it is possible to bind a fanout exchange with a direct exchange. The messages in the fanout exchange also get delivered to the direct exchange. The direct exchange in turn load balances this over the subscribers in the direct exchange.

## Monitoring

Due to the current used and purchased software, monitoring of Docker system metrics can best be done with PRTG, which offers a specialized Docker sensor. When adding the Docker sensor PRTG requires a secure connection to Docker by means of SSL. This will create an encrypted link between Docker and PRTG. This also means that the Docker machine can be monitoring externally over IP.

Setting up Docker to only accept SSL connections does add a level of complexity due to all other tools or software that is being used with Docker should also utilize SSL. An alternative to PRTG is a tool that can be run from Docker and opens the metrics from external sources instead of connecting to Docker directly with software or a tool.

When choosing for a different monitoring tool that can be run inside Docker, the availability matching the Docker container environment will differ. The number of tools for monitoring Docker and Linux containers greatly exceeds the offer for Windows containers. An alternative most recommend is Nagios.



At the current scale of Semso, custom made dynamic monitoring software can also be an option. Since the platform is not developed, a monitoring microservice can be added to display microservice calls.

RabbitMQ offers a website to monitor the data flow of queues and exchanges, these metrics can also be collected by PRTG, but Docker will add a new level of complexity in form of configurations in this scenario.

When Docker is deployed off-site, on a Windows machine, Netsh commands are used in the shell to configure the TCP/IP protocol. This can be used to reroute incoming traffic on a certain port to reroute internally to the IP RabbitMQ uses to display the metrics. On Linux the same can be achieved with “*iptables*”.

Both RabbitMQ and PRTG are recommended by Semso and are familiar with way metrics are displayed.

### *Logging*

At current scale, elaborate logging tools are not recommended aside from a database that collects all operations done by the platform. When implementing the output to the database, it is important to already implement the logging standards.

Since the platform will be run inside Docker, the containers log to a default location using the default output. When declaring the docker-compose file, it is possible to alter the logging driver to the suit 3<sup>rd</sup> party software needs, for example JSON. Docker itself offers output for existing software syslog and splunk etc.

### *Unit Testing*

Microsoft Visual Studio offers a unit testing module to test the microservice functionality. When the tests are written and run, Visual Studio will generate a report based on the success of these tests.

### *Platform*

Based on the gathered knowledge, the platform interaction sequences can be seen in Figure 12. In the diagram the arrows indicate the exchange with the message content shown between brackets. As soon as the polling service or web hook activate they will leave a message in the “imported” exchange with a message containing the relative data. All exchanges will use the company client name as their routing key. This operation and all future operations are logged.

The formatting service will consume the messages in the “imported” exchange and save the data ordered to a local database. When a user requests an export in a predefined format holding selected information, the exporting service will send a message to the “format request” exchange with the selected specifications as message. On this message the formatting service will reply in the “formatted” exchange with the formatted data for the exporting service to make an export of.

All logs can be used to visualize the flow of the platform when making a monitoring microservice.

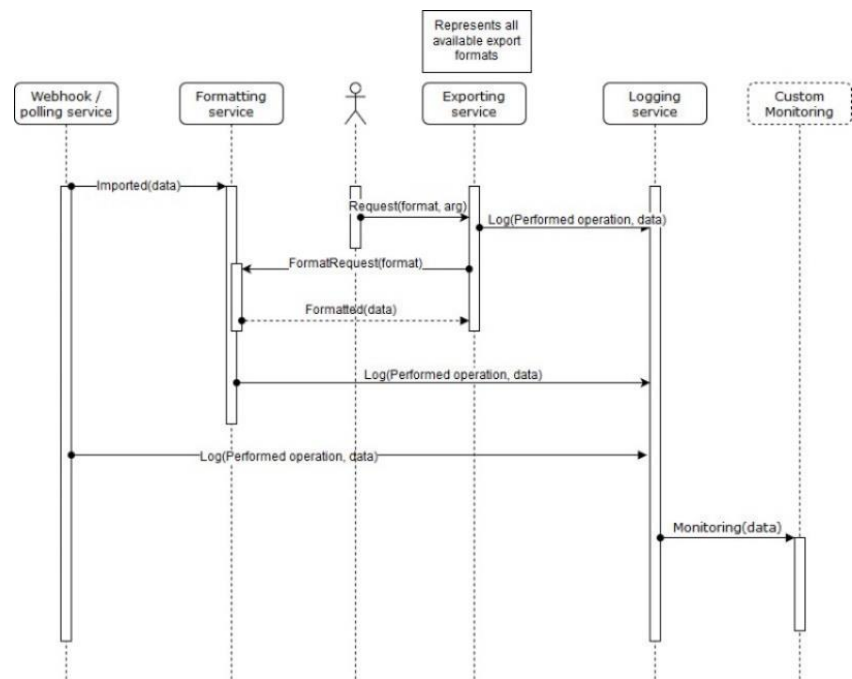


Figure 12. Suggested Sequence Diagram

A blueprint of the platform can be seen in Figure 13 where the arrows represent in which exchange they leave the message and the text above the arrow indicating what the message contains. With this setup new clients can quickly be added to the platform when many of the characteristics are similar, for example the data source is similar or the export format.

When the exchanges are setup as seen in the diagram, the rate of external requests and internal data transfer can be seen by monitoring the exchanges. All microservices, the formatting service in particular have their own databases and can only access data from different microservices through a request.

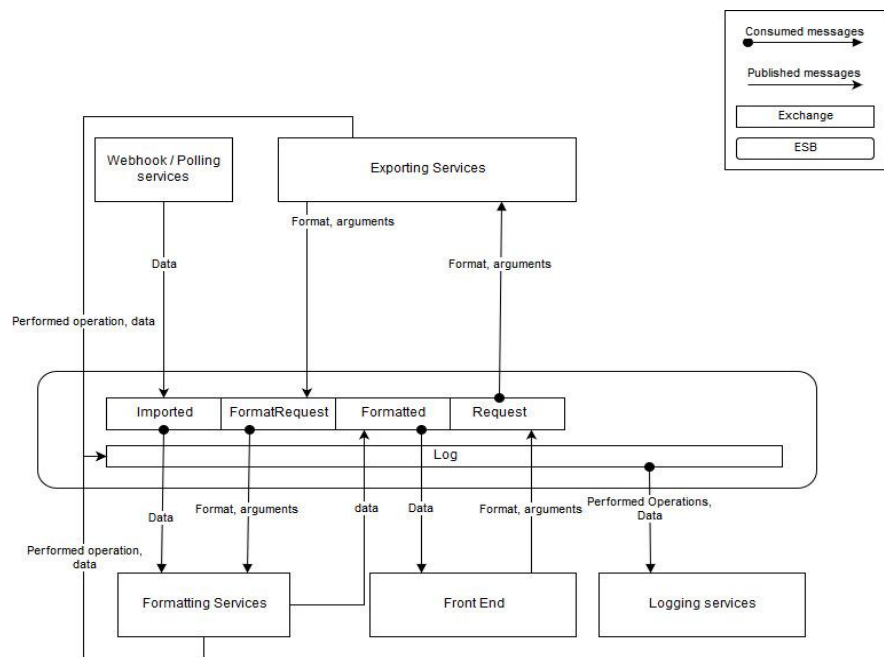


Figure 13. Blueprint of platform

## *Measurement of success*

With the design laid out, the terms of satisfaction can be answered as followed:

### ***Modular Integration:***

When adding or removing microservices with similar or different functionality the existing microservices did not change behavior.

### ***Monitorable data flow:***

The monitoring interface to inspect rate of transfer has already been used before by the employees of Semso.

### ***Existing library use:***

The limited amount of 3<sup>rd</sup> party software needed to achieve the goals has actually been a boon in contradiction to a multitude of libraries. The following software is needed to build this platform:

- Docker
- RabbitMQ
- Visual Studio
  - RabbitMQ Nuget package.
  - Git
- PRTG

### ***Unit testing:***

Microsoft Visual Studio offers a unit testing module as well as version control.

### ***Integration testing:***

When a microservice set to perform the integration test, all microservices in the same exchange will react and acknowledge the added microservice without any interruption in the platform.

### ***Continuous deployment:***

The problems concerning continuous deployment have been circumvented with the use of containerized software.

### ***Externally monitorable***

The Docker system can be approached from any external source when the internal IP of the system running Docker reroutes external calls to the internal Docker IP.

### ***Microservice configuration***

Configuration of the microservices that do the polling for example are hard coded in an external file which can be given as argument to the microservice. Creating a microservice that acts as a client host holding client information is still being discussed.

### ***Hot-pluggable***

One of the benefits of creating a microservice oriented architecture is that it offers hot-pluggable and modularity as architecture feature.

## *8. Further studies*

To follow up on the knowledge acquired from the assignment an interesting follow up is how this can be applied to the gaming industry. The similarities between programming a platform like the one

described in this document and writing a network game are plenty. A practical application could be a microservice oriented game engine.

## 9. Products

### Load Balancing, Hot plugging and modularity

To prove that RabbitMQ can handle the load balancing in an exchange to consumers of the same type, a proof of concept(PoC) is written in C#. In Figure 14 the theoretic representation can be seen of load balancing.

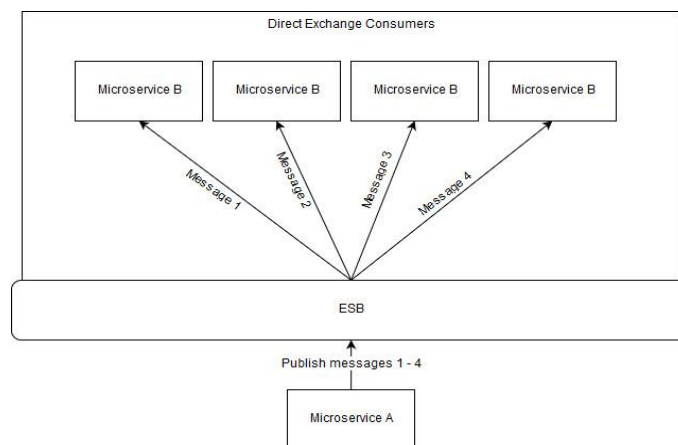


Figure 14. Theoretic representation PoC.

Seen in Figure 15 is the result of 1 publisher and 3 identical consumers. In the figure it is shown how the top window publishes “hello world” with an ID. This ID is to see which consumer handles the message.

In this figure can be seen how the messages from the publisher are balanced between 3 consumers. The 3 consumers are represented by the bottom 3 windows.

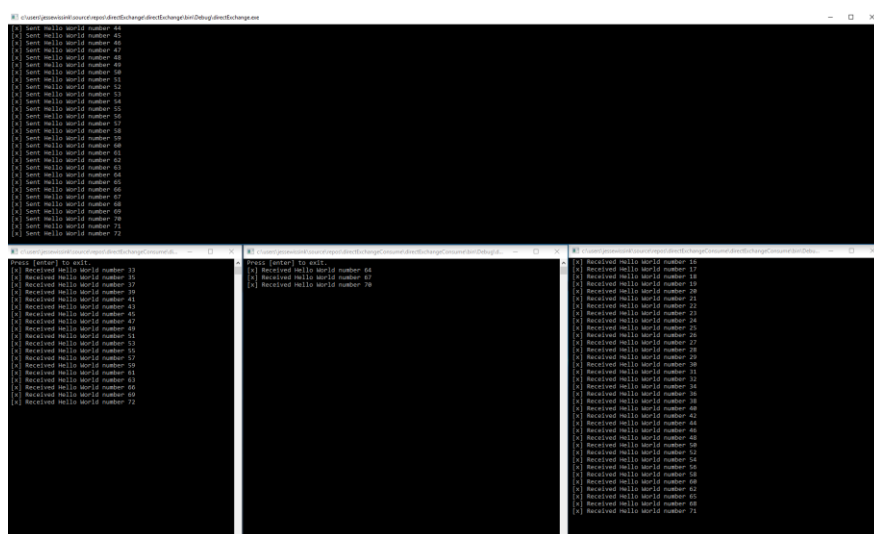


Figure 15. Load balancing PoC

To extend on this, there are certain messages that might utilize the same exchange but differ in consumers. In RabbitMQ this is called routing. This can be achieved with routing keys. A visualization of the routing process can be seen in Figure 16, the PoC result can be seen in Figure 17.

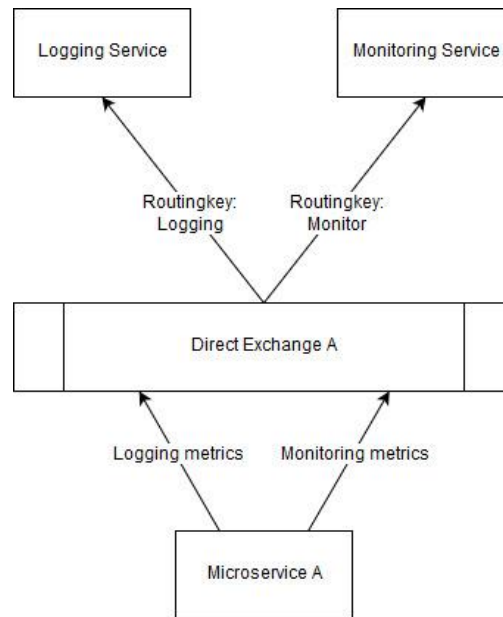


Figure 16. Routing visualization

```

C:\Users\jess@source\repos\directExchange\directExchange\bin\Debug\directExchange.exe
=====
Exchange: Sempo
RoutingKey: Dynamically changing
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
Previous:
[13:47] Sempo | Logging | Hello World number 412 |
[13:47] Sempo | Monitor | Hello World number 413 |
[13:47] Sempo | Logging | Hello World number 414 |
[13:47] Sempo | Monitor | Hello World number 415 |
[13:47] Sempo | Logging | Hello World number 416 |
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
[13:47] Sempo | Monitor | Hello World number 417 |
=====

Opdrachtprompt - directExchangeConsumer.exe "Logging"
=====
Exchange: Sempo
RoutingKey: Logging
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
Previous:
[13:47] Sempo | Logging | Hello World number 406 |
[13:47] Sempo | Logging | Hello World number 408 |
[13:47] Sempo | Logging | Hello World number 410 |
[13:47] Sempo | Logging | Hello World number 412 |
[13:47] Sempo | Logging | Hello World number 414 |
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
[13:47] Sempo | Logging | Hello World number 416 |
=====

Opdrachtprompt - directExchangeConsumer.exe "Monitor"
=====
Exchange: Sempo
RoutingKey: Monitor
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
Previous:
[13:47] Sempo | Monitor | Hello World number 407 |
[13:47] Sempo | Monitor | Hello World number 409 |
[13:47] Sempo | Monitor | Hello World number 411 |
[13:47] Sempo | Monitor | Hello World number 413 |
[13:47] Sempo | Monitor | Hello World number 415 |
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
[13:47] Sempo | Monitor | Hello World number 417 |
=====

```

Figure 17. PoC Routing

```

C:\Users\jess@source\repos\directExchange\directExchange\bin\Debug\directExchange.exe
=====
Exchange: Sempo
RoutingKey: Dynamically changing
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
Previous:
[14:03] Sempo | Logging | Hello World number 890 |
[14:03] Sempo | Monitor | Hello World number 891 |
[14:03] Sempo | Logging | Hello World number 892 |
[14:03] Sempo | Monitor | Hello World number 893 |
[14:03] Sempo | Logging | Hello World number 894 |
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
[14:03] Sempo | Monitor | Hello World number 895 |
=====

Opdrachtprompt - directExchangeConsumer.exe "Logging"
=====
Exchange: Sempo
RoutingKey: Logging
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
Previous:
[14:03] Sempo | Logging | Hello World number 884 |
[14:03] Sempo | Logging | Hello World number 886 |
[14:03] Sempo | Logging | Hello World number 888 |
[14:03] Sempo | Logging | Hello World number 890 |
[14:03] Sempo | Logging | Hello World number 892 |
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
[14:03] Sempo | Logging | Hello World number 894 |
=====

Opdrachtprompt - directExchangeConsumer.exe "Logging"
=====
Exchange: Sempo
RoutingKey: Logging
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
Previous:
[14:03] Sempo | Logging | Hello World number 884 |
[14:03] Sempo | Logging | Hello World number 886 |
[14:03] Sempo | Logging | Hello World number 888 |
[14:03] Sempo | Logging | Hello World number 890 |
[14:03] Sempo | Logging | Hello World number 892 |
#Time#|Exchange|RoutingKey|Message|
-----|-----|-----|-----|
[14:03] Sempo | Logging | Hello World number 894 |
=====

```

Figure 18. PoC Direct Fanout

When multiple consumers subscribe to the same routing key, in a direct exchange, RabbitMQ will fanout the messages, the result of this PoC can be seen in Figure 18.

## Continuous Delivery

With Docker as delivery platform, the pipeline from deployment to delivery is hugely shortened. This due to the possibility of disregarding operating specific variables. The docker delivery pipeline introduces complexity on its own regarding configurations and commands.

With the rise of containerized software, and Docker being open-source project from Microsoft, the IDE “*Microsoft Visual Studio 2017*” offers automatic delivery to Docker.

When making a .NET core console application, Docker support can be added to the current project. This command seen in Figure 19 will add a Dockerfile and a Docker-compose.yml file.

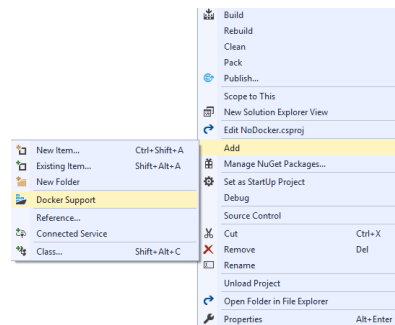


Figure 19. Add docker support

The generated files can be seen in Figure 20 and Figure 21. The first figure displays a compose file. In this file developers can define which services they want build and booted and in what order.

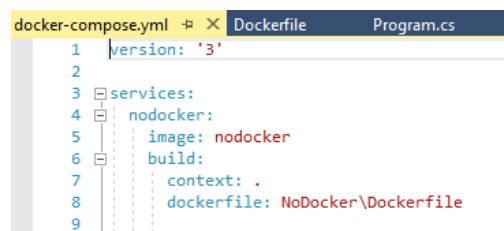
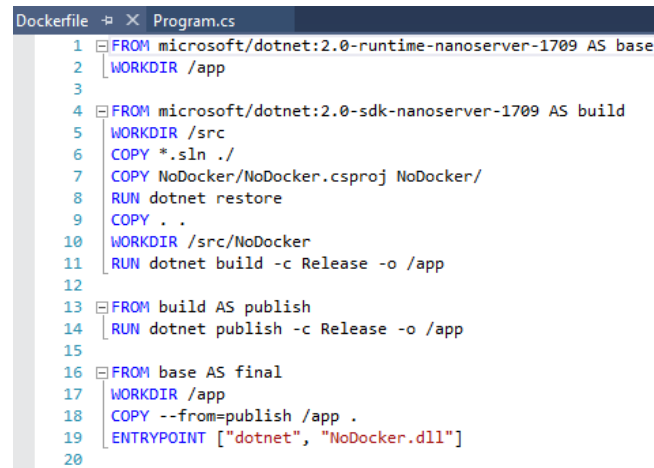


Figure 20. Docker-compose example

The version relates to the docker-compose version. The application that have to “spin-up” (boot) are declared under services. In this example, the application is named “*nodocker*”, in Docker, the image will save as defined after “*image:*”. Followed by the information on how to build the project. The instructions on how to build the console application is defined in the Dockerfile, seen in Figure 21.

In Figure 21 the build options for the console application are defined. First it defines the base image, which is the image needed to run the application. In this example it’s the .NET runtime environment.



```

1 FROM microsoft/dotnet:2.0-runtime-nanoserver-1709 AS base
2 WORKDIR /app
3
4 FROM microsoft/dotnet:2.0-sdk-nanoserver-1709 AS build
5 WORKDIR /src
6 COPY *.sln ./
7 COPY NoDocker/NoDocker.csproj NoDocker/
8 RUN dotnet restore
9 COPY . .
10 WORKDIR /src/NoDocker
11 RUN dotnet build -c Release -o /app
12
13 FROM build AS publish
14 RUN dotnet publish -c Release -o /app
15
16 FROM base AS final
17 WORKDIR /app
18 COPY --from=publish /app .
19 ENTRYPOINT ["dotnet", "NoDocker.dll"]
20

```

Figure 21. Dockerfile Example

### Microservice configuration

By making use of arguments when launching a microservice, the decision can be made to also pass the location of the configuration file and exchange to use. Continuing with the “nodocker” example, arguments would be passed to the microservice by adding “CMD”.

```

COPY ./config.cfg ./
ENTRYPOINT ["dotnet", "NoDocker.dll"]
CMD ["exchangeName queueName config.cfg"]

```

When not using docker-compose, the arguments could be given in the following way:

```
docker run nodocker exchangeName, queueName, config.cfg
```

An abstract working representation of the platform can be downloaded and run with docker from

<https://github.com/jwissink/platform/blob/master/docker-compose.yml>

To run the platform, install Docker for the appropriate OS and set the container type to windows. When this is complete navigate to the downloaded folder with the shell or batch. Inside the folder containing the docker-compose.yml file type

```
docker-compose up -d -build
```

This command will download and install the entire platform on to docker, including dependencies.

The source code for the docker containers can be found at:

<https://hub.docker.com/u/jwissink/>

## 10. Appendix I – Bibliography

Bogard, J. (2016, January 16). *Avoiding microservice megadisasters*. Retrieved from <https://www.youtube.com/watch?v=gfh-VCTwMw8>

- Cerarcn. (2017, April 4). *Microservices-based architectures enable contious delivery/deployment*. Retrieved from Eventuate.IO: <https://blog.eventuate.io/2017/01/04/the-microservice-architecture-is-a-means-to-an-end-enabling-continuous-deliverydeployment/>
- Docker. (2017, November 14). *Build, Ship and run any app, anywhere*. Retrieved from Docker: <https://www.docker.com/>
- Dynatrace. (2017, November 14). *Deliver unrivaled digital experiences*. Retrieved from Dynatrace: <https://www.dynatrace.com/>
- Ford, N. (2017, March 17). Building Microservice Architectures. Vienna, Austria. Retrieved from <https://www.youtube.com/watch?v=pjN7CaGPFB4>
- Fowler, M. (2014, January 1). *GOTO 2014 Microservices*. Retrieved from <https://www.youtube.com/watch?v=wgdBVIX9ifA>
- GoCD. (2017, November 14). *Simplify Continuous Delivery*. Retrieved from GoCD: <https://www.gocd.org/>
- Grafana Labs. (2017, November 28). *About Grafana*. Retrieved from Grafana Labs: <https://grafana.com/>
- IBM. (2012, Januari 1). *Connecting services through an enterprise service bus*. Retrieved from IBM Knowledge center: [https://www.ibm.com/support/knowledgecenter/SSFTDH\\_8.0.0/com.ibm.wbpm.main.doc/topics/cwesb\\_esb.html](https://www.ibm.com/support/knowledgecenter/SSFTDH_8.0.0/com.ibm.wbpm.main.doc/topics/cwesb_esb.html)
- Jenkins. (2017, November 14). *Build great things at any scale*. Retrieved from Jenkins: <https://jenkins.io/>
- Kavis, M. (2014, April 15). Nagios is not a monitoring strategy. United States of America.
- Lewis, J. (2014, 10 29). Microservices. (S. Radio, Interviewer) Retrieved from [http://www.se-radio.net/?powerpress\\_pinw=1550-podcast](http://www.se-radio.net/?powerpress_pinw=1550-podcast)
- Logalyze. (2017, November 28). *Logalyze. Search, find, analyze*. Retrieved from Major Features: <http://www.logalyze.com/product/major-features>
- Nagios. (2017, November 14). *The industry standard in IT infrastructure Monitoring*. Retrieved from Nagios: <https://www.nagios.com/>
- Netflix, R. M.-D. (2016, September 2). *Microservices at netflix scale: principles, tradeoffs & lessons learned*. Retrieved from <https://www.youtube.com/watch?v=57UK46qfBLY>
- Newman, S. (2016). *Building Microservices - Designing fine-grained systems*. United States of America, Sebastopol, California: O'Reilly Media.
- OWASP. (2017, October 20). *Logging cheat sheet*. Retrieved from OWASP: [https://www.owasp.org/index.php/Logging\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Logging_Cheat_Sheet)
- Paessler. (2017, November 14). *The network monitoring company*. Retrieved from Paessler: <https://www.paessler.com>



- Pivotal. (2017, November 30). *AMQP concepts*. Retrieved from RabbitMQ: <https://www.rabbitmq.com/tutorials/amqp-concepts.html>
- Pivotal. (2017, November 14). *Most widely deployed open source message broker*. Retrieved from RabbitMQ: <https://www.rabbitmq.com/>
- Prince, S. (2016, June 28). We thought we were doing continuous delivery and then.. New York City, New York, United States of America.
- Richardson, C. (2015, February 24). *Pattern: Microservice Architecture*. Retrieved from Microservices.io: <http://microservices.io/patterns/microservices.html>
- Richardson, C. (2017, November 30). *The Scale Cube*. Retrieved from Microservices.io: <http://microservices.io/articles/scalecube.html>
- Splunk. (2017, November 10). *Splunk*. Retrieved from Splunk: <https://www.splunk.com/>
- Vasters, C. (2015, October 5). Introduction to AMQP 1.0. Microsoft Corporation, Washington, United States of America. Retrieved from <https://www.youtube.com/watch?v=ODpeldUdClc>
- Wähner, K. (2013, April 2). *Choosing the right ESB for your integration needs*. Retrieved from InfoQ: <https://www.infoq.com/articles/ESB-Integration>
- Wähner, K. (2016, March 8). Microservices: Death of the enterprise service bus. Drottningholm, Sweden. Retrieved from <https://www.youtube.com/watch?v=fITFdDU5L9w>
- XebiaLabs. (2017, November 14). *DevOps at Enterprise Scale*. Retrieved from XebiaLabs: <https://xebialabs.com/>