NTNU
Norwegian University of
Science and Technology

Message Service

Authors
Manuel Jesús Bravo García
Tom Roar Furunes
Tomasz Rudowski

Bachelor in Software Engineering
20 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

16.05.2018

Supervisor Frode Haug

Sammendrag av Bacheloroppgaven

Tittel:	**Meldingstjeneste**
Dato:	16.05.2018
Deltakere:	Manuel Jesús Bravo García Tom Roar Furunes Tomasz Rudowski
Veiledere:	Frode Haug
Oppdragsgiver:	Headit AS
Kontaktperson:	Rune Kollstrøm, rune.kollstrom@headit.no, 62510052
Nøkkelord:	Bachelor, Message, API, Java, Spring, Angular
Antall sider:	146
Antall vedlegg:	11
Tilgjengelighet:	Åpen

Sammendrag:

Siden menneskets opprinnelse har utveksling av informasjon vært tilgjengelig. Nye kommunikasjonsmetoder oppsto samtidig med teknologisk fremgang, for å svare på den stadig økende etterspørselen. Selv om mange forskjellige kommunikasjonskanaler er tilgjengelige, er leveringstid fortsatt en bekymring. Hvordan øke effektiviteten for levering av meldinger mellom en sender og mottaker? Dette prosjektet er dedikert til å løse problemet med å sende meldinger ved å tilby en Internett-tjeneste for registrerte applikasjoner. Tjenesten kan integreres med alle applikasjoner som er koblet til Internett; den tilbyr konfigurasjon av både sender og mottaker, og inkluderer støtte for programmerbar automatisk sending uten interaksjon fra brukeren av applikasjonen, dette kan føre til at direktemeldinger sendes uten forsinkelse når en bestemt betingelse blir oppfylt i mottaker-applikasjonen.

Summary of Graduate Project

Title:	**Message Service**
Date:	16.05.2018
Authors:	Manuel Jesús Bravo García Tom Roar Furunes Tomasz Rudowski
Supervisor:	Frode Haug
Employer:	Headit AS
Contact Person:	Rune Kollstrøm, rune.kollstrom@headit.no, 62510052
Keywords:	Thesis, Bachelor, Message, API, Java, Spring, Angular
Pages:	146
Attachments:	11
Availability:	Open

Abstract:

Exchanging of information has been present in human history since the very beginning. New communication methods emerged along the technology progress to respond the growing demands. Although many different communication channels are available, there is still a concern of the delivery time. How to increase efficiency in delivering of a message from a sender to a recipient? This project is dedicated to solving a problem of automatic sending of a message by providing an Internet service for registered applications. The service could be integrated into all applications connected to Internet; it provides the configuration of both sender and receiver and includes a support for programmable automatic sending without an interaction from the user of the application, which could allow an instant message sending without delay when certain conditions occur in the sender application.

Preface

This project has been developed for the IT-consulting company Headit AS (Hamar, Norway). We would like to thank the entire company for the resources they provided us.

We would like to thank Bjørn Tore Wiken and Ronny Kristiansen (Headit representatives) for their support, inspiring ideas and the feedback they gave us. They have guided us along the project kindly and with patience.

Finally, a big thanks to our project supervisor Frode Haug for the help, orientation and good feedback we got from him to develop this project.

Contents

List of Figures

Listings

1 Introduction

The scope, boundaries and target audience of the Message Service project are introduced first. The second part of this chapter briefly describes the realisation of the project and the structure of this report.

1.1 Scope

1.1.1 Field of Study

Along history, humans have had the need for sharing with others thoughts, feelings or what have happened around us. The way this need of communication is fulfilled varies depending on the kind of information, whom this information is transmitted to, where is the receiver and the technology that is available at the moment; sometimes we can just speak directly with another person to transmit the information. In some cases, we need that the information is received by people that are far away from us physically and/or in time; then it is important that the message remains unchanged regardless of time. Time itself might also be a concern; some messages may need immediate, or at least quick, delivery.

Over time we have developed many different communication systems: cave paintings, hieroglyphs, smoke signals, messenger pigeons... And more recently we use phones, emails, SMS or social networks to exchange information with others.

1.1.2 Delimitation

The exchange of information is not exclusive to human beings; machines can also exchange information between themselves or with humans.

The information is very valuable for companies in order to know and serve the needs of theirs customers in an efficient way. If a company has access to the right information at the right time and is able to understand it, then the company will be able to use its resources effectively and this can constitute a competitive advantage.

In this project, we will develop a system that makes possible to exchange information between different applications, systems and users that are part of a company. This information will be sent using different channels e.g. SMS or email and a copy of this communication must be kept. The system must be able to add new communication channels in the future and in addition it must offer a communication interface so it can be used by many different systems/applications.

1.1.3 Project Description

Headit AS is an IT-consulting company located in Hamar. They want to develop a messaging hub (MaaS, Message as a Service) that collects, interprets and reacts to the information that is generated within a company. The service will work as a stand-alone application and it must be possible to adapt the service to the concrete needs of each of their customers. It should be possible to install it on customers infrastructure or using Cloud technology (SaaS).

The system will:

- Receive and interpret the messages and route them via different channels,
- Save all the messages in such a way that every message is linked with the application and person (role or username) that sent it.
- Offer a dashboard where the users have the messages (they have sent before) overview, and where they can define: which applications can use the messaging service and which channels will be available for them. It will be possible to send messages to other users using this dashboard.
- Be possible to configure against external services.

Figure 1 presents an overview of the requested functionality. MaaS receives messages from the applications and it forwards them to the final recipient; a web application will be used to configure which applications will have permission to use the system.

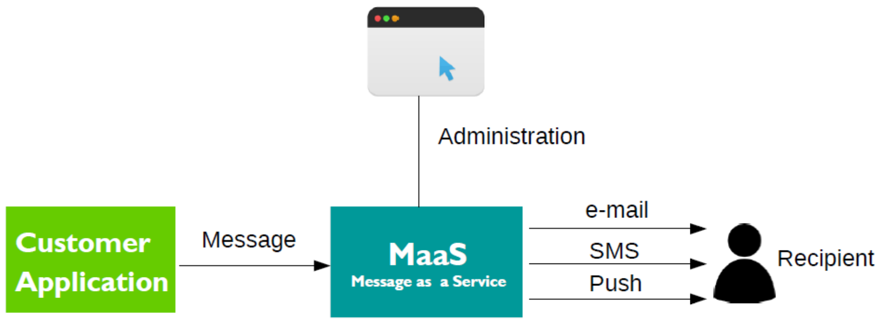


Figure 1: MaaS service overview

1.2 Project choice

When we discussed the choice of the subject for our bachelor thesis we noticed how MaaS could be useful as an integrated component of many service systems; regardless of actors involved a messaging hub could be opened for communication between application, services, system components or in a wider perspective IoT (Internet of Things). The expanding potential of the system convinced us that it would be valuable experience for us to work on it; although complexity and dependency on external service indicated that significant amount of time would be consumed by research, learning new technology and integration issues.

1.3 Audience Target

The MaaS is an application which aims to collect, interpret and react to the events that may occur in the different business processes of a company.

1.3.1 Application Target

The main target of this application are organisations that want to make their processes more effective. The MaaS will allow these companies to manage the information that is generated in their business processes in an efficient way. At the same time these organisations will be able to react faster to certain events; as a result, they could save time, money and improve the customer experience.

1.3.2 Report Target

The person who reads this document should have a certain background and knowledge in software development e.g. any person that studies or teaches software engineering. That is why some terms and definitions will be omitted.

1.4 Background and Experience

We are software engineering students with experience in development of components that could become parts of the system, but this has been the first time we have been supposed to put all of them together; from planning and design to implementation and production. Additionally to working with creating of a complete system we would get experience with new technologies and integration of the components of the system, both between each other and with external services we would use.

1.5 Methodology and Project Organisation

Manuel Bravo, Tom Roar Furunes and Tomasz Rudowski have been working on the bachelor thesis during a spring semester 2018. Supervisor of the project is Frode Haug.

Already during the first meeting with Headit we agreed to use an agile Scrum methodology (details in section 8.1.3). Stakeholders representatives are Ronny Kristiansen (Product Owner) and Bjørn Tore Wiken.

We took into account that disagreements could occur during the project. We made and signed a document with group rules we could use in case that any disagreement or dissatisfaction occurred.

Realisation of the project includes following activities:

- Planning. Dedicated to create a Project Plan (appendix F), define and confirm system context, boundaries, interaction and behavioral models.
- Research. Focused on new technologies and integration possibilities between internal and external systems, noticed both while planning and during development.
- Architecture and Design. Creation of a model of the system (including noticed patterns) as a base for implementation and adjusting it when necessary for new technologies/solutions implemented later.
- Programming. Implementation and testing of planned system architecture.

1.6 Report Organisation

The report is divided into eight chapters (chapters following this one are listed below) that describe work progress on the project and explain choices made. In appendix to the report it is possible to find additional data concerning work progress; those documents are project plan, meeting logs, API specification, user stories and scenarios. Report has been written using LaTeX in ShareLaTeX [1]. Diagrams were created using an online application draw.io [2]

[1]https://www.sharelatex.com/
[2]https://www.draw.io/

- The second chapter of this report presents requirements of the system and the elicitation process.
- The third chapter presents the model of the system along with the presentation of the process towards establishing it.
- The fourth chapter describes the development process from the chosen methodology perspective; it includes work overview and tools used for development.
- The fifth chapter is focused on programming tasks and challenges during implementation.
- The sixth chapter is dedicated to preparations made for the deployment of the system.
- The seventh chapter presents how the system has been tested.
- In last chapter choices made during development process are discussed with focus on what have been developed. It contains also the evaluation of the project and the proposal for the potential expanding of system functionality. The report is concluded with the summary of the results of the project.

There are two types of references in this report:

- Footnotes - those are used when a technology or a concept is named, usually refer to online resources, that might contain additional information (for the report reader) about the concept (URLs have been validated by the submission date - 16.05.2018)
- Bibliography - resources that have been used during the development of the project.

2 Requirements

In this chapter we present the requirements elicitation process, which has been carried out in order to discover the requisites that our application must fulfil. The last part of the chapter is dedicated to the initial product backlog that was created according to Scrum methodology.

2.1 Requirements Elicitation Process

To elicit the different requirements, we used an iterative process described by Sommerville [1]. Initially we started extracting requisites from an introductory project description we got from Headit. Then we continued discovering more requirements through the questions to the stakeholders representatives and the discussions we had with them. We used stories and scenarios to get a better understanding about the application we were going to develop. Finally all the requirements, gathered during a planning phase, were formally written down and validated. Since we decided to use Scrum methodology, we created an initial Product Backlog based on those requirements; additional requisites, that appeared later as we developed the system were discussed and included in it.

Here we will further discuss the outcomes of the elicitation process as it follows:

- Actors that have interaction with MaaS.
- General system functional requirements. They describe what is expected that the system should do.
- Non-functional requirements that further define system constraints and stakeholders expectations that don't affect functionality directly.
- Stories and scenarios overview, based on created business case.
- Detailed requirements specification that includes both user requirements and system requirements.
- Use case diagram from a perspective of different actors.
- Sequence diagrams that show process flow in two cases: registering an application and sending a message.

2.2 Actors

We identified following actors that could interact with the system:

- MaaS Admin - a superuser that administrates MaaS.
- App Admin - an administrator for a group of applications using MaaS.
- App - an application registered at MaaS, eligible to use the service.
- App User - an user of an application connected to MaaS.
- Receiver - an application, hardware (e.g. mobile phone) or other interface the outgoing message is send to, from the MaaS server.
- Recipient - an user of a Receiver.

Those names will be referred to in this report; in this chapter for presenting requirements, use cases and system sequence diagrams.

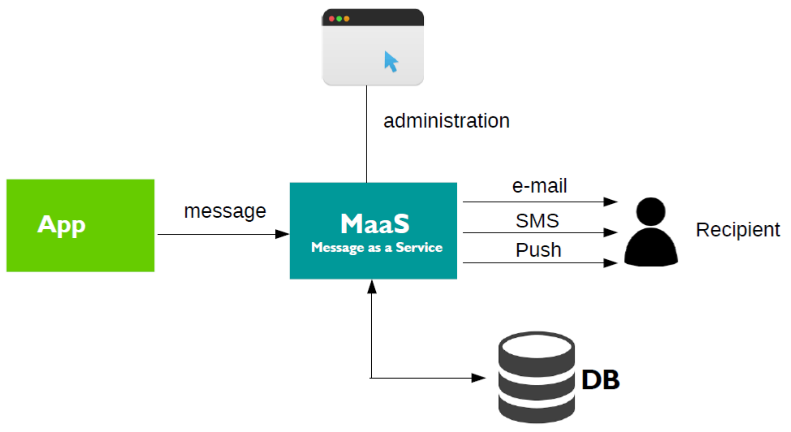


Figure 2: System functionality

2.3 System Functional Requirements

MaaS will work as a message hub; its primary functionality is to receive and forward messages from authorised sources. It is expected that those authorised sources could be added, removed and configured by an user, who has an admin role in this context. Additional control should be given through message log.

2.4 System Non-functional Requirements

2.4.1 Non-functional Product Requirements

Although it has been decided, that changes that need to be done to an App registered at MaaS are out of scope for the project (appendix J.2) and will be fulfilled by a third part we should address some of them; concerning automatic sending of message to Recipients integrated in the registered application, support should be considered and realised in form of API specification to help integration with MaaS. API specification can be found in appendix E.

2.4.2 Organisational Requirements

Development requirements

Since the source code is expected to be maintained, integrated and enhanced by stakeholders there have been named development requirements to ensure fluent transition.

- Make system easy to maintain and expand to potentially new receiver channels.
- Use of well known frameworks currently used by stakeholders. As examples: Spring[1] for backend, and Angular[2] for frontend.
- Chosen database technology should be wide used and well supported.
- Design system as a stand-alone application concerning cloud deployment using

[1]https://spring.io
[2]https://angular.io

Docker[3] technology.

- Integrate system with existing OAuth[4] service, like e.g. Keycloak[5].

2.5 Stories and Scenarios

A My Custom Suit business case (appendix C.1) has been created in cooperation with stakeholders to present a possible usage of the MaaS system in practice. My Custom Suit is a hypothetical company, that owns a chain of shops and cooperate with independent tailors to provide custom-made suits to the customers. MaaS system would potentially increase effectiveness of information flow between shops and tailors and between shops and customers; and reduce storage costs for ready parts of the suit.

2.6 Detailed Requirements Specification

2.6.1 User and System Requirements

Headit expectations and technical limitations, gather during meetings and presented by them in initial task description were used to create a list of User Requirements; by further analysis of those requirements we have classified and organised them. Our goal was to find detailed System Requirements, that could be prioritise through negotiations with Headit during development process.

1. App User can send a message to another application (from App)
   - Implement outgoing channel that can send a message to an App (registered application).
   - Receivers (applications) registered as an App can read incoming messages through REST API.
   - An application shares both Sender and Receiver properties to support two-way communication.
2. App Admin has an application overview and possibility to configure App that can use the messaging system
   - Store configured applications in database.
   - Create new application.
   - View a list of applications.
   - Edit application.
   - Delete application.
3. App Admin wants the configuration of the Apps permanently recorded.
   - Store configuration of the Apps in database.
4. App Admin can configure each App separately
   - Edit App properties in admin panel (web-application).
   - Edit App properties in App.
5. App Admin has sent messages overview for Apps that are under his/her administration. Log view.

[3]https://www.docker.com
[4]https://oauth.net
[5]https://www.keycloak.org

- Store messages in database.
- View a list of messages.

6. App Admin can send a message to one or more Apps/App Users (from admin panel)
   - Interface for crafting messages and sending in admin panel (For App Admins).
7. Only App registered by App Admin can send a message to MaaS using a sett of defined channels.
   - API that can receive messages.
   - Able to send messages.
   - Control permissions of the App sending the message.
   - Generate API key for each registered app.
   - Generated API key should be tied to App Admin and App.
8. Messages received from App will be forwarded to different channels defined by App Admin
   - Receive messages formatted to a standard that fits log.
   - Log all messages that are forwarded.
   - Receive messages formatted to a standard that fits interface of outgoing channel component.
   - Implement different outgoing channels for sending.
9. App is allowed to view own configuration at MaaS.
   - API that can send stored configuration of an App.
10. MaaS Admin can configure access of App Admin users of MaaS.
    - Create new App Admin.
    - View list av App Admins.
    - Edit App Admin profile.
    - Delete App Admin.
    - Store App Admins in database.
11. Recipient wants to receive a message delivered by preferred method. (external channels like email, mobile application etc)
    - Implement outgoing channel for sending email.
    - Implement outgoing channel for sending via webhook.

2.6.2 Use case

Use case diagram (figure 3) presents usage of the system from a perspective of different user types. Here only human actors are represented to show interaction with system functionality without (omitted in the picture) middle-ware i.e. App for App User and Receiver for Recipient.

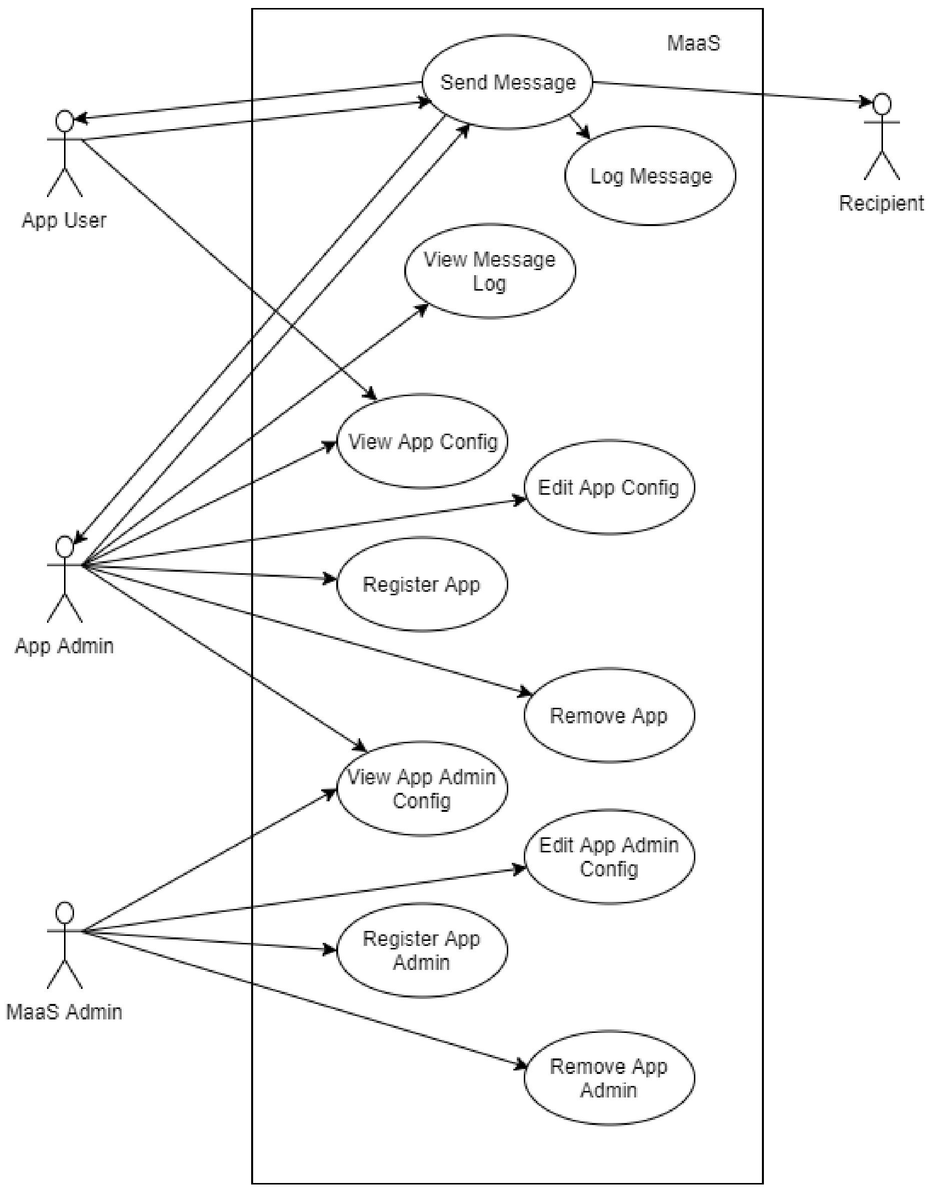


Figure 3: Use case from the perspective of different user types

Additionally sequence diagrams present the usage of the system in two cases. One of them shows an example of usage in application configuration context, here (figure 4) adding a new application that will be allowed to use MaaS system; notice that obtained access key is generated by MaaS but need to be registered (stored) in the App. Primary MaaS functionality i.e. sending a message is shown on figure 5. The access key is used to verify that App is allowed to send a message through MaaS; log entry is created and message is forwarded towards recipient.

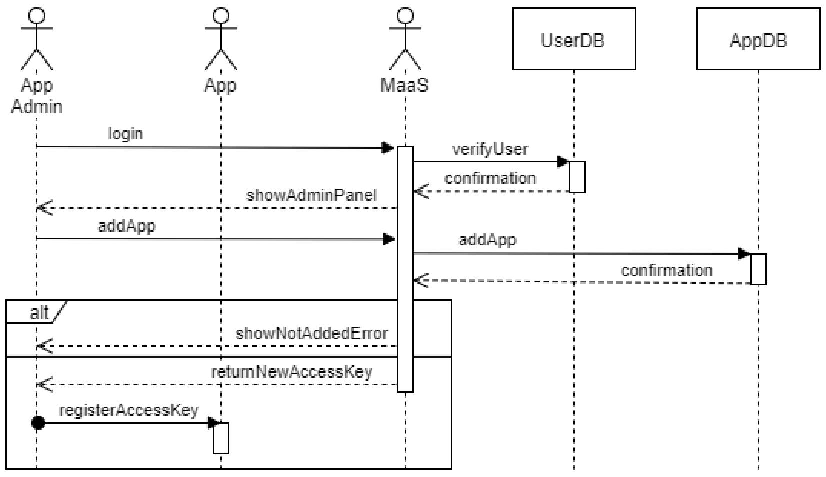


Figure 4: Sequence diagram for registration of App

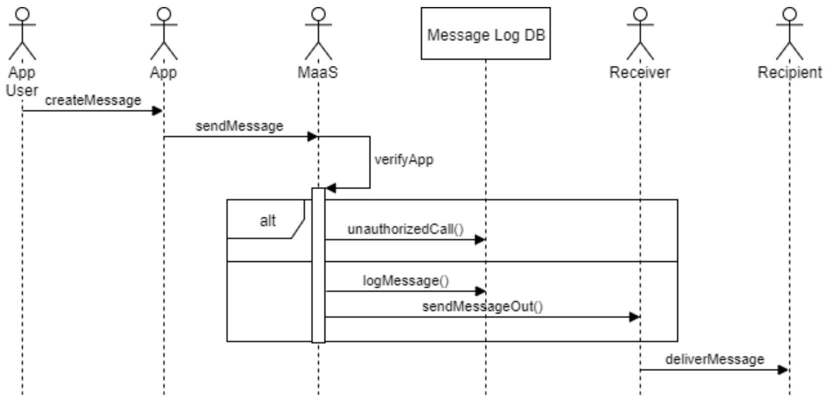


Figure 5: Sequence diagram for sending a message from App User to Recipient

2.6.3 Software Requirements Document

Requirements presented in section 2.6.1 have been published in Confluence space belonging to Headit.

2.7 Requirements Validation

Requirements were validated after initial (planning) phase of the project. Changes have been accepted after each modification with concern to the system as a whole, i.e. how changes influence the final form of MaaS.

2.8 Requirements Change

Considering project deadline and complexity, requirements changes were discussed and expected to come already in the beginning. Using agile methodology it has been possible to manage changes during sprint meetings.

2.8.1 Requirements Management Planning

Confluence has been used to manage changes in requirements. Using version control mechanisms the previous states of requirement list have been checked and referred to.

2.8.2 Requirements Change Management

There have been room for discussion about current project progress on every sprint meeting. If problems have been noticed, they have been analysed and changes have been applied if possible.

2.9 Product Backlog

We used Jira to manage the Product Backlog and Sprints. We started with organising group of requirements into Epics. Each Epic has included Stories based on User Requirements and smaller Tasks based on System Requirements, all those were presented previously in section 2.6.1.

Later we divided each Task into smaller sub-tasks which could be handled within a day of work, more about the Product Backlog and Sprints during development in chapter 4.

3 Technical Design

In this chapter we present a technical design of the project. In the beginning the system is considered as one element placed in environment as in the initial project description; models presented after the first, general view are more detailed and concern functionality derived from requirements presented in previous chapter. After the final, most detailed model of system architecture was established, we considered patterns that could be applied; those patterns are presented in a section following general system modelling. Next section is dedicated to database design and shows the model as a result of the analysis of the data needed to be stored in the system. Following sections describe in details components of the system based on chosen technology and patterns; technologies not known previously are explained along with practical application of them in order to prepare a base for implementation.

3.1 Modelling of System Components

The model of the system was created based on requirements analysis. Starting from the most general approach to show the context of the system from an external perspective (figure 6), through more detailed view to extract basic system components (figure 7), towards the final model in the third step with detailed components dependencies (including integrated OAuth service - Keycloak, described in details in section 5.1.1) and system boundaries (figure 8). Goal for this analysis was to confirm understanding of the system as the whole, without going into details about implementation; the model was accepted by stakeholders.

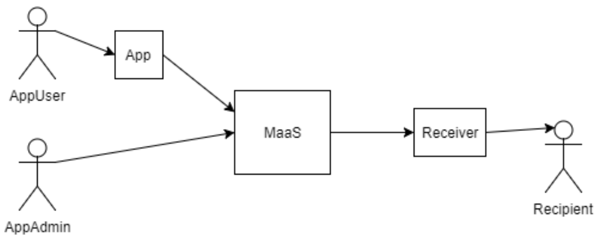


Figure 6: System model (step 1)

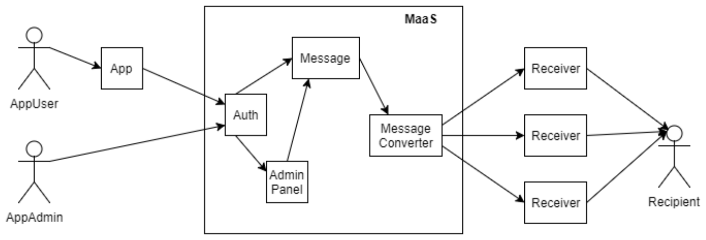


Figure 7: System model (step 2)

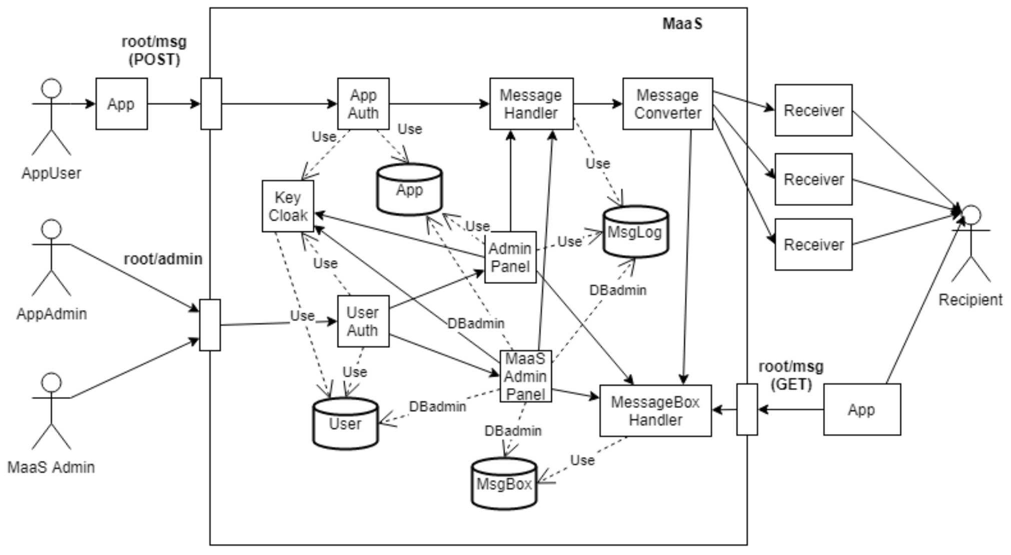


Figure 8: System model (step 3)

3.2 Database Modelling

We used database design methodology described by Connolly and Begg [2]. We started with a conceptual database design to create representation of data appearing in the MaaS system. The next step was to build a logical design, on that step it was necessary to decide which data model would be implemented. The last step, physical design, concerns a concrete implementation of database.

3.2.1 Conceptual Database Design

This section presents the first phase of the methodology; on that point no assumptions regarding implementation constraints were made. We used a top-down approach to build a database model by identifying first main entities and relationships between them and then including more details in a similar way to presented previously approach to modelling of system components. The first main entities were Message and Actor (in sender role); later the model was extended to include required Log and various Channels for Messages and a Recipient that could be an internal or external actor. Finally Actor and Recipient entities were enhanced using specialisation, to represent distinct variants of entities, and Message Postbox was introduced to note one, special, outgoing channel i.e. internal message. Results are presented using Entity-Relationship diagram (figure 9); this is a general first-cut view without attributes associated with entities; sender and internal receiver of a message represented here as a separate entities for clear view would be implemented as a one, Actor entity. We defined simple and composite attributes along with their domains and decided on primary keys for each entity and confirm the model with stakeholders (details of the tables presented later in section 5.4.5).

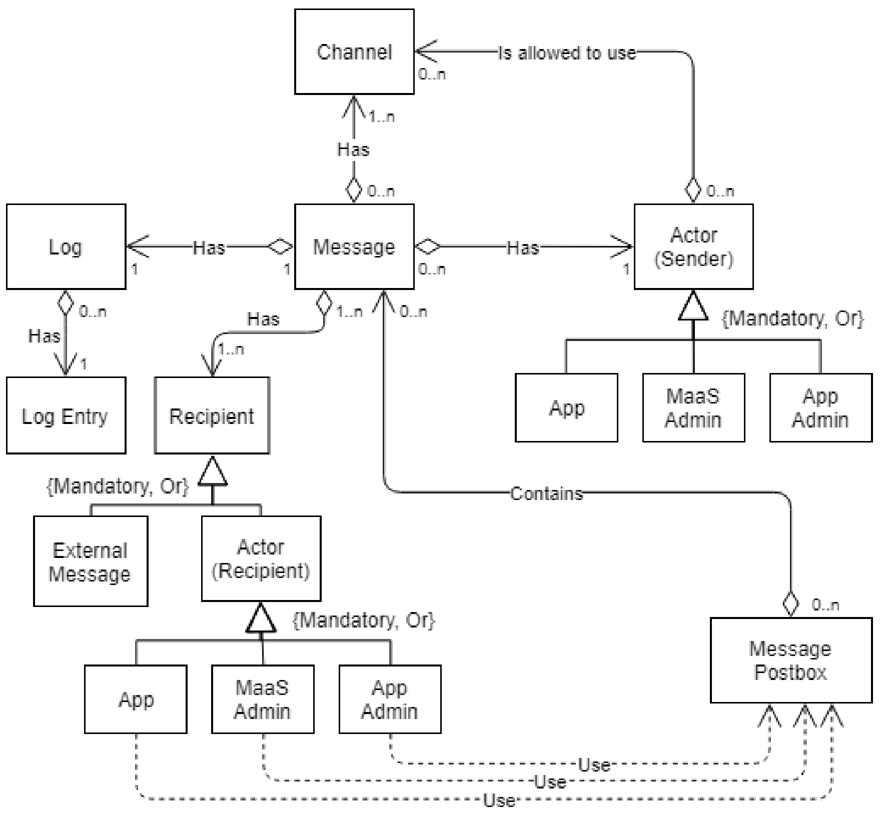


Figure 9: Entity-Relationship model for message sending

3.2.2 Logical Database Design

We decided to use relational database model, mainly based on previous experience; we evaluated new technologies we needed to used in the entire project and concluded that we might not had enough time to additional research in this field. We created the logical data model based on the choice. Superclass/subclass relations for Actor and Recipient entities were defined as mandatory and disjoint. Relations were validated using normalisation and integrity constraints included. More details about relations in section 5.4.3.

3.2.3 Physical Database Design

The third phase of database design process describes how to implement the database in a concrete chosen technology. We decided to use MariaDB[1] server as a relational database. The reason for it was that it is an open-source and well documented technology as requested by stakeholders (section 2.4.2).

To manage and communicate with the database, Java Persistence API (JPA)[2] is used, which is implemented by Hibernate[3]. JPA facilitates relational databases through Java code and every table is a Java class with corresponding fields and relations [3].

A class annotated with @Entity[4] represents a table in the database; each property in

[1]https://mariadb.org
[2]https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html
[3]http://hibernate.org/orm/
[4]https://docs.oracle.com/javaee/6/api/javax/persistence/Entity.html

this class that is annotated with @Column is a column in this table (that allows customisation of tables). It is also possible to create Many-to-Many and One-to-Many relations using annotations.

We wanted to get familiar with the new technologies while implementing backend in Java, and the ones chosen allow to implement Java classes that represent underlying physical database realisation; the benefit of using this technology is that it is possible to replace the database server if required, without changes in Java code.

3.3 Architecture and Design Patterns

After confirmation of the model of the system (figure 8), following patterns were noticed and decided to implement.

3.3.1 Client-Server

Two main features of MaaS i.e sending a message and configuration of access to message server are supposed to be done remotely through REST API or/and browser (web-application). Application processing in configuration context consists of user rights check (i.e. control if operation is allowed for the user requesting it), interaction with database and authorisation server. Application processing in message sending context consists also of access control (based on rights of the requesting application) and forwarding message toward outgoing channel. Presentation functionality could be implemented on the client side.

Based on foregoing description choice was made to propose a client-server architecture (figure 10) with a thin-client [1].

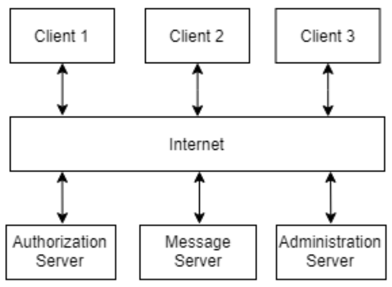


Figure 10: Client-server architecture

3.3.2 Backend

General Layered Model

Figure 11 presents layered architecture model of MaaS backend [4]. The top layer are controllers awaiting HTTP requests and responding accordingly to the results of further request processing. The second layer are services that implement MaaS application business logic. The third layer consist of functionality that allows access to the database layer that lies at the bottom. Layers are described later in details in section 3.4.

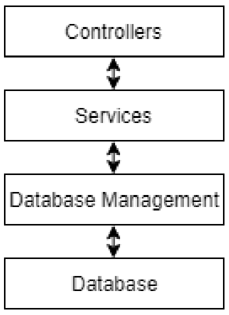


Figure 11: Layered architecture model for backend

Channel Factory

Factory Method Design pattern [5] has been proposed to be used to implement processing of accepted messages. A Message Service creates Message object based on HTTP Requests. Channel Factory creates a concrete object based on message type. The Message object is forwarded to it. Each implementation of Channel interface corresponds with one defined message type and is responsible for converting and sending message (also to an external system). Message processing model that includes a Factory Method pattern is presented on figure 12.

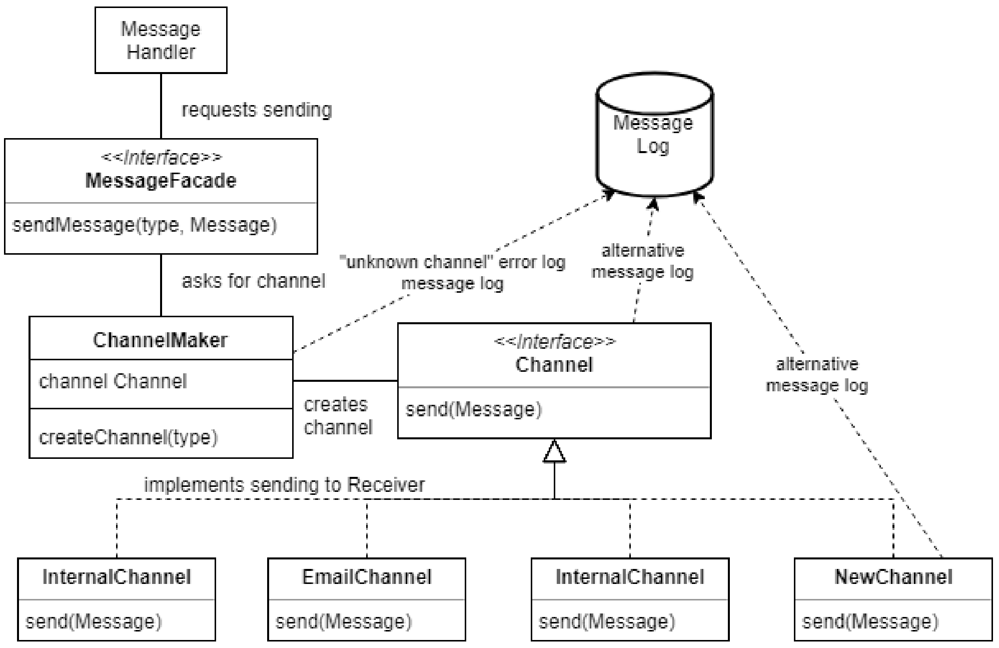


Figure 12: Message processing model for outgoing channels

3.4 Details of Backend Layers

3.4.1 Controllers Layer

Controllers in the application are classes which consist of methods (handlers) that represent each API endpoint (See figure 8 on page 13).

The backend was developed using the Spring Framework[5] to help creating the controllers, such that only the controllers itself needs to be implemented, not the underlying technology.

Spring Boot[6] was used further to achieve this. Spring Boot creates a stand-alone Spring application with a Tomcat web-server[7] which is automatically configured.

Java annotations[8] are used to configure the Spring application and communicate with the Spring framework; these annotations provide data to the Spring framework which can be used to configure and serve the application; annotations are pre-fixed with @.

To create controllers, classes with the @RestController annotation are used to inform the Spring Framework that the class should act as a Rest controller, which means that every public function annotated with @RequestMapping (or equivalent) in this class is a handler for an API endpoint. @RequestMapping is an annotation telling which endpoint (requests) this function should handle. [6].

MaaS application has two classes that are Rest controllers:

- **AdminController** - for handling requests to /admin/api endpoint (e.g. admin web-application).
- **MessageController** - for handling requests to /msg/api endpoint (e.g. message sending, fetching inbox etc).

Figure 13 presents an example of class diagram for one controller and services related to it (two first layers of backend architecture). More on how this is implemented in the application later in section 5.4.1.

3.4.2 Services Layer

The services in the application are classes where the business logic is handled. These classes are annotated with @Service [9].

A service class is (by default) a Singleton class that holds all the functionality for business logic. A service could be used by other services or the controllers. To access a service it is added to the class trough Auto-wiring with the @Autowired annotation which is a Dependency Injection to inject this object into the desired class (in MaaS context - the controllers or other services) [6, 7]. Figure 13 presents an example of a class diagram for one controller and services related to it (two first layers of backend architecture). More on how this is implemented in MaaS later in section 5.4.2.

There are nine services, that handle CRUD operations for:

- Actor
- Actor Type

[5]https://spring.io
[6]https://projects.spring.io/spring-boot/
[7]http://tomcat.apache.org
[8]https://docs.oracle.com/javase/tutorial/java/annotations/
[9]https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html

- Channel
- Domain
- User
- Log
- Message
- Message Type
- Permission

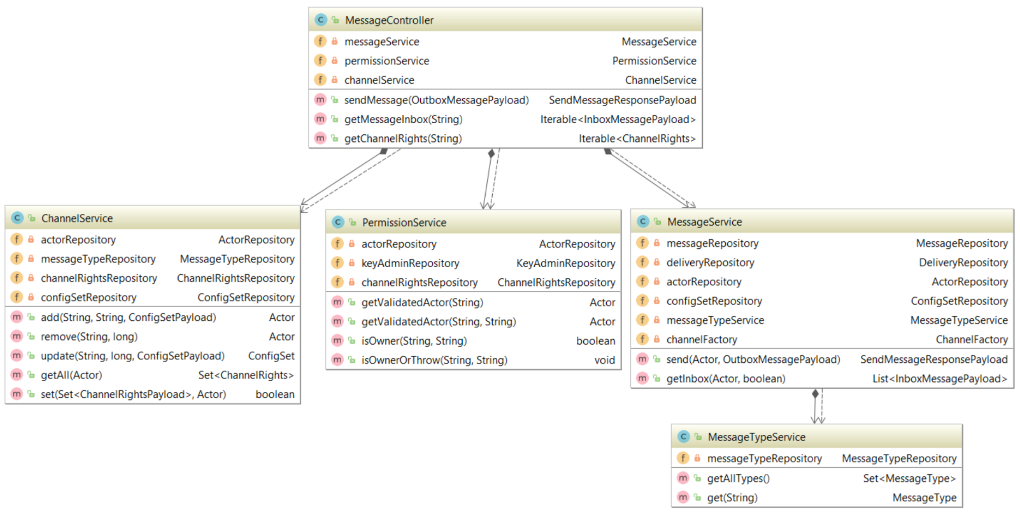

MessageController
messageService MessageService
permissionService PermissionService
channelService ChannelService
sendMessage(OutboxMessagePayload) SendMessageResponsePayload
getMessageInbox(String) Iterable<InboxMessagePayload>
getChannelRights(String) Iterable<ChannelRights>
ChannelService
actorRepository ActorRepository
messageTypeRepository MessageTypeRepository
channelRightsRepository ChannelRightsRepository
configSetRepository ConfigSetRepository
add(String, String, ConfigSetPayload) Actor
remove(String, long) Actor
update(String, long, ConfigSetPayload) ConfigSet
getAll(Actor) Set<ChannelRights>
set(Set<ChannelRightsPayload>, Actor) boolean
PermissionService
actorRepository ActorRepository
keyAdminRepository KeyAdminRepository
channelRightsRepository ChannelRightsRepository
getValidatedActor(String) Actor
getValidatedActor(String, String) Actor
isOwner(String, String) boolean
isOwnerOrThrow(String, String) void
MessageService
messageRepository MessageRepository
deliveryRepository DeliveryRepository
actorRepository ActorRepository
configSetRepository ConfigSetRepository
messageTypeService MessageTypeService
channelFactory ChannelFactory
send(Actor, OutboxMessagePayload) SendMessageResponsePayload
getInbox(Actor, boolean) List<InboxMessagePayload>
MessageTypeService
messageTypeRepository MessageTypeRepository
getAllTypes() Set<MessageType>
get(String) MessageType

Figure 13: Example of class diagram for one controller and services related to it, IntelliJ

3.4.3 Database Management Layer

For database management the Data Access Object Pattern [8] is used.

Data Access Object Pattern

Services contact Repositories that implements CrudRepository[10] interface from Spring framework; the interface provides CRUD functionality on a repository of a specific Entity type. Transfer Object (as described by Oracle [8]) is represented on a figure 14 by an Entity object, that is used here as an abstract representation of the underlying database structure.

[10]https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

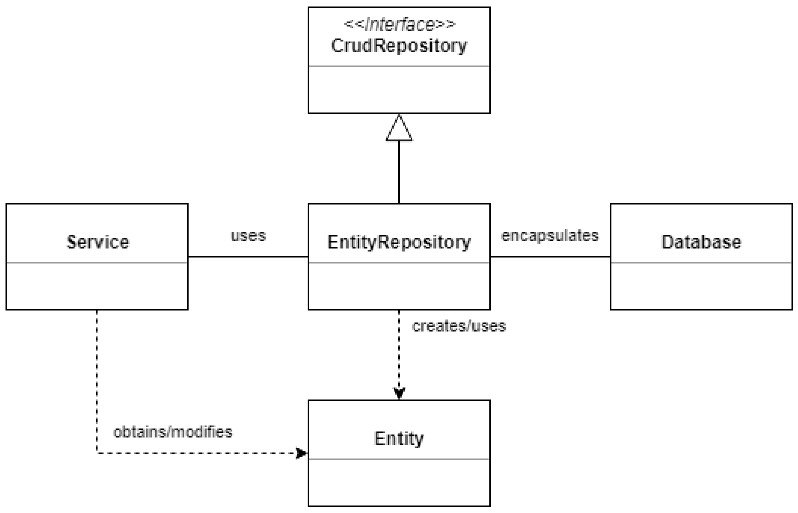


Figure 14: Data Access Object pattern

Entities with corresponding repositories in our application are:

- Actor (Sender and Recipient)
- Actor Type
- Admin
- Channel Rights (Channel an Actor is allowed to use)
- Config Set
- Delivery
- Domain
- External Recipient
- Log
- Message
- Message Type

This is described in details in implementation chapter (section 5.4.3).

3.4.4 Database Layer

This layer would be implemented using a MariaDB server integrated with the other components of the system (integration details in chapter 6).

3.5 Frontend

Based on requirements from stakeholders (section 2.4.2) it was decided to use Angular 5[11] to develop the frontend, i.e. a web-application to be used as an admin panel (for App Admins) to administrate Apps as described in requirements chapter 2.

Angular 5 is a framework for developing client side web-applications using TypeScript[12] and HTML. The framework provides a set of TypeScript libraries that are imported into the application and allow creation of Components and Services [9].

[11]https://angular.io
[12]https://www.typescriptlang.org

In addition we used Bootstrap[13] to simplify creation of HTML.

3.5.1 TypeScript

TypeScript is similar to JavaScript concerning the syntax and semantics, except that it doesn't run directly in a browser, but it compiles to JavaScript so the browsers can run the code.

3.5.2 Components

Components do typically consists of a HTML document, CSS and a TypeScript file. The TypeScript file consists of a class decorated with @Component, in this class all logic for that specific component is handled. The HTML and CSS files include only the code for that specific component, but it is posible to create more complex structures (one component can include another component). The web-application consists of components created for each layout element that has a specific purpose, such as top navigation bar, different forms, lists etc. More on how this was implemented is described in chapter 5.

3.5.3 Services

A service is a class that holds specific functions and variables that other components can use. The services are included in the components by using Dependency Injection, defining the variable in the components constructor; then this service is available in that component, similar to auto wiring used by Spring described earlier. There are three services in MaaS web-application:

- **MaasService** - controls all communication with the backend API.
- **DataService** - holds data needed by components.
- **AlertService** - handles error responses from backend API.

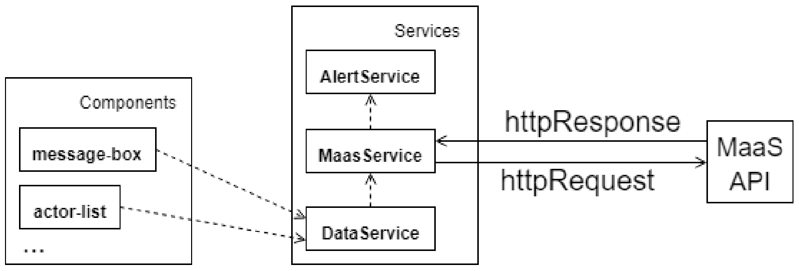


Figure 15: Relations between components and services in frontend model

Figure 15 shows relations between components (two example components) and services. Detailed implementation of services described in section 5.3.

[13]https://getbootstrap.com

4 Development Process

Details of the development process will be explained in this chapter. First, we will describe how we decided to share the tasks and the responsibilities along the project. So we will go into details about what has be done during every of the nine sprints, the project was divided into. Finally we will present the different tools that have been used in this project.

4.1 Methodology

Due to the complexity of the project, our experience and the project deadline, we decided to nine sprints with a duration of two weeks each (60 working hours per sprint for each of us).

4.1.1 Roles and Responsibilities

All group members; Manuel Jesús Bravo García, Tom Roar Furunes and Tomasz Rudowski are part of the development team and share other roles related to Scrum methodology. Since we had similar experience and our goal was to learn, we decided to share a Scrum Master responsibilities by using a combination of “Rotating Scrum Master” and “Part Time Scrum Master” [10]. Each group member had to continue to work as a part of development team (with lower workload) when it was his time to take a Scrum Master role.

In addition to the roles that arise from the chosen methodology, we decided to give the additional responsibilities to group members considering project progress management and reporting (more details in the Project Plan - appendix F.3.1). These responsibilities were meant to ensure that all the tasks were done and all requirements fulfilled. We wanted to learn as much as it was possible and all the group members have helped each other and have contributed in the different parts and tasks of the project.

4.1.2 Methodology in Practice

We started each working day with a Scrum meeting at the campus, where most of the work was done; during that meeting we discussed tasks that were recently done and to do in the upcoming working day. After we chose the tasks and evaluated time needed, we assigned them to group members (depending on task the work could be done alone or by two group members together - pair programming). At the end of the day we discussed progress and current status; if some task were too time consuming or difficult we could decide on some extra individual work later at home.

Every Sprint started with a Planning Meeting and ended with Review Meeting with the representatives of the stakeholders in Hamar.

Due to practical reasons both Review (of previous Sprint) and Planning (of the next one) Meetings took place at the same day. Therefor stakeholders representatives were present when we discussed and evaluated the previous Sprint (Retrospective Meeting); this gave us opportunity to share not only results but also working methods with them.

4.2 Work Progress

4.2.1 Sprint 1, 10.01.2018 - 23.01.2018

The first sprint was mainly dedicated to creation of the Project Plan (appendix F) and requirements elicitation process. We additionally conducted research concerning new technologies like Spring and Keycloak. Headit provided us an access to Jira, Confluence and Bitbucket space.

4.2.2 Sprint 2, 24.01.2018 - 06.02.2018

Initial set of requirements was created and validated. Architecture and Design proposals (general model and patterns) were accepted and included in the Project Plan. Project Plan was approved by supervisor and submitted. We followed tutorials for chosen technologies (Spring, Keycloak), and run code examples to get more familiar with them.

4.2.3 Sprint 3, 07.02.2018 - 20.02.2018

We created Product Backlog in Jira. Discussed details concerning interpretation of requirements. Main programming activity was Keycloak integration. We started with setting up Docker environment for Keycloak and database instances. Later we created Spring project and implemented controllers that fetched data from the Keycloak instance. We started an Angular project (basic frontend web-interface connected to backend API). Access to admin API was secured by Keycloak. We created controllers for message sending API and secured them with Access Keys generated in database (UUID). Proposed solutions were accepted and integration level satisfactory on that point.

4.2.4 Sprint 4, 21.02.2018 - 06.03.2018

In the beginning of the fourth sprint we worked with refactoring of code base; request from stakeholders included splitting the project into two smaller modules keeping them in separate repositories (developed independently). We rebuilt the project structure and transferred the code base into two repositories; one for Angular project (frontend), and the second one for Spring project (backend) which included also additional configuration files for integrated services (Keycloak, database).
Programming tasks:

- Backend. Added CRUD functionality for Actor and supporting ActorType. Decided and implemented Actor removal as deactivation to keep data consistence and possibility to activate/deactivate access for defined periods of time. Added Log functionality to the first step of message handling.
- Frontend. Created following views in Admin Panel: Log, Actor List, Actor Details; added functionality of sending message and create Actor as modal components.

4.2.5 Sprint 5, 07.03.2018 - 20.03.2018

Main focus in this sprint was to implement exchanging of internal messages between registered applications. It was realised in two steps:

- The first one included implementation of API and support for data flow on backend and message inbox interface on frontend (new page on Admin Panel).
- The second step extended configuration functionality, so application could be defined based on channel access rights and individual configuration of messages. Ad-

ditionally we introduced a Current Actor concept to Admin Panel to keep context in all available views and added delivery status to Log view.

4.2.6 Sprint 6, 21.03.2018 - 10.04.2018

During this sprint we implemented error and exception handling common for entire backend and refactored controllers and services to keep clear separation between layers and by this allowing easier unit tests for business logic. Additionally we started to create an API specification document and reviewed all code base to ensure code quality.
Another achievement for the sprint was to follow message send to another channel (Slack webhook) all the way through the system from sender to recipient; this feature was tested against mobile and desktop receiver.

4.2.7 Sprint 7, 11.04.2018 - 22.04.2018

Main focus in this sprint was on writing the final report from the project. Changes to code base included handling of issues noticed during tests and Review Meeting; as for new features an email channel was implemented according to newest requirements, the second enhancement was adding App and App Admins domains.

4.2.8 Sprint 8, 25.04.2018 - 08.05.2018

As in the previous Sprint, the main work was put into the project report. Programming tasks concerned code quality and enhancing the way the error responses from backend API are presented in Admin Panel.

4.2.9 Sprint 9, 09.05.2018 - 22.05.2018

Development work focused mainly on code quality and additional testing. Last corrections to the final report was made.

The final submission of this report is in the middle of this Sprint. We decided, that all Sprints should have the same workload (as presented in the Gantt diagram for the project work progress in appendix B); therefor the remaining week after the report submission will be dedicated to additional work around the project. We plan to prepare the presentation of the project for both NTNU and stakeholders (demonstration date hasn't been decided yet); concerning practical issues, we need to manage resources used during development, among them data storage, accounts, software licences etc. We are also prepared that this time might also be used for potential issues considering source code (transition, maintenance) not noticed before submission.

4.3 Tools

Different tools have been used during the project. Here we will review how they have been used.

4.3.1 Project Management

Both Confluence[1] and Jira[2] access were provided by Headit. Confluence is a communication platform we used to share information related with the project. We use it e.g. to share business cases, meeting and decision logs, and sprint review retrospectives. Jira

[1]https://www.atlassian.com/software/confluence
[2]https://www.atlassian.com/software/jira

is a project managing tool. It was utilised to manage the Product Backlog, organise the tasks in epics and stories. In every sprint planning meeting we assigned which tasks we were going to develop during that sprint by moving them into the Sprint Backlog.

4.3.2 Version Control

Git is an open source version control system developed by Linus Torvalds [11]. We used GitBash[3] daily e.g. when tasks got done or when we fixed bugs. We made use of branches when we developed new features, in case of refactoring and when we wanted to try out new technologies and solutions preserving the master branch. Stable and accepted versions were merged with the master branch in both backend and frontend repositories.

4.3.3 Communication

Slack[4] is a communication tool that the development group used daily to share practical information, ask and resolve questions in a fast and effective way. Confluence and Jira were used as communication tools as well. We employed them to share "formal information" related to the project status and progress e.g. meeting and decision logs (Confluence) or task assignments (Jira). Emails and Skype have been used as well.

4.3.4 Development Environment

Regarding software development, we set up and made use of an environment composed of the following tools:

- IntelliJ IDEA[5]: An Integrated Development Environment (IDE). The frontend and the backend were developed as separated projects. We imported the frontend project (as a module in the backend project) in order to be able to work with both projects simultaneously.
- Docker[6]: Under the development process, we made use of docker; several containers where we run the applications and servers we needed: MariaDB, PhpMyAdmin, Keycloak.
- Maven[7]: Is an open source tool that simplifies the building process of an application. We used it to integrate libraries such as Spring, Keycloak and JPA.
- Npm[8]: Package manager for the libraries used by the frontend.
- AngularCli[9]: Was employed to simplify the creation of the frontend project, creating components and services. It was used as well to test the application.
- Postman[10]: Used to send HTTP requests to MaaS under development, replacement for a custom mock sender application.

4.3.5 Resource Storage

The frontend and backend projects were developed separately and stored in two different repositories. GoogleDrive were use in order to share and save documents and pictures from group meetings.

[3]https://git-scm.com/
[4]https://slack.com/
[5]https://www.jetbrains.com/idea/
[6]https://www.docker.com/what-docker
[7]https://maven.apache.org/
[8]https://docs.npmjs.com/getting-started/what-is-npm
[9]https://cli.angular.io/
[10]https://www.getpostman.com

4.3.6 Report and Documentation

The project's report has been written using LaTeX, Latex Table Generator and BibTeX Editor. Diagrams like class diagrams, system diagrams were created using draw.io. MS Project were utilised to create the Gantt diagram for the project.

5 Implementation

In this chapter we present a practical implementation of the MaaS system. The first section of the chapter is dedicated to integrated services that were used along the whole implementation process. The next section consists of files and packages overview for both backend and frontend. The implementation process and details for applied solutions are presented using two process flows.

The first of them is a registration of a new application in MaaS. Application management is presented starting from an Admin Panel (frontend implementation) through backend API towards changes in the database.

The second process, chosen to present the implementation solutions, is sending a message through MaaS. The process is followed from an App perspective (including assumptions necessary to use MaaS); later the message is processed through all stages, i.e. receiving a request, App verification, creating a Message object (the representation of a message in MaaS), finding a correct outgoing channel, sending a message to it and finally creating a response to the sender.

5.1 Integrated Services

MaaS includes three integrated services. One of them is MariaDB server running in a Docker instance (implementation details described later in chapter 6); the second one is phpMyAdmin, that we used only during implementation for database management purposes (data control); the third one is Keycloak. All those services were running inside Docker containers on each developer's machine.

5.1.1 Keycloak

Keycloak is an open source identity and access management system. It offers a single sign-on system i.e. once the users are logged in Keycloak, they will be logged in all applications that use this instance of Keycloak avoiding several logins. Keycloak offers an Admin Console where it is possible to manage users and applications [12]. It is in this Admin Console where an administrator of MaaS (Maas Admin) can add users; then the users can be assigned to one of the domains (this functionality is implemented in Admin Panel frontend); an user with an assigned domain (App Admin) can manage applications (App).

Keycloak can have several realms; one of them is Master, which is the default one, with access to this realm, a user has all rights to manage this Keycloak instance (e.g. create new and manage other realms, add users, etc.). Since it is recommended to use another (than Master) realm to manage users, applications etc, a “MaaS” realm is created [13].

MaaS system must be able to authenticate users through Keycloak, to achieve that, a “client” (a representation of an application secured with Keycloak) called “maas-app” is created in Keycloak Admin Console. There are also two pre-defined roles created; “user” and "maas_admin". The "user" role represents a App Admin, and the "maas_admin" role represents a MaaS Admin.

In order to secure applications, Keycloak offers "client adapters" (libraries) for several and well known platforms and programming languages, among then Spring Boot [14]. The backend of the MaaS System makes use of the Keycloak library through the KeyCloakUser service. This service uses the Keycloak adapter for Spring Boot; it is able to identify the user that is trying to access MaaS and find out if the user is logged in or not.

```java
@Service
public class KeyCloakUserService implements UserService {
    ...
    Keycloak keycloak;
    @Value("MaaS")
    private String realm;
    private HttpServletRequest request;
    ...
    @Override
    public CurrentUser getCurrentUser() {
        if (!isLoggedIn()) {
            return null;
        }

        KeycloakPrincipal<KeycloakSecurityContext> kp =
        (KeycloakPrincipal<KeycloakSecurityContext>) request.getUserPrincipal();
        String userId = kp.getName();
        final AccessToken token = kp.getKeycloakSecurityContext().getToken();
        String username = token.getPreferredUsername();
        String firstname = token.getGivenName();
        String lastname = token.getFamilyName();
        String email = token.getEmail();
        Set<String> roles = token.getRealmAccess().getRoles();
        return new CurrentUser(userId, username, firstname, lastname, email, roles);
    }
    @Override
    public boolean isLoggedIn() {
        Principal principal = request.getUserPrincipal();
        return principal != null && principal instanceof KeycloakPrincipal;
    }
...
}
```

Listing 5.1: Extract of the KeyCloakUser Service

In listing 5.1 we use the KeycloakPrincipal object to get information about the logged in user from Keycloak, such as "username", "email" and "roles".

5.2 Files and Package Structure

In this section the organisation and the structure of the source code is presented. Frontend and backend are placed in two separate repositories; project configuration files are omitted.

5.2.1 Backend

Package structure for backend is presented on fig. 16. Controllers, Services and Database Management layers has been placed in separate packages. Each channel is to be implemented as a sub-package, which should contain all necessary information to interpret message metadata and convert it to a format acceptable by an external (receiver) service. The "payload" package consists of classes that help to format JSON payload in the context of HTTP Requests and Responses; except for those channel specific payloads, which would be implemented in a channel sub-package. The "error" package contains classes responsible for the exception and error handling.

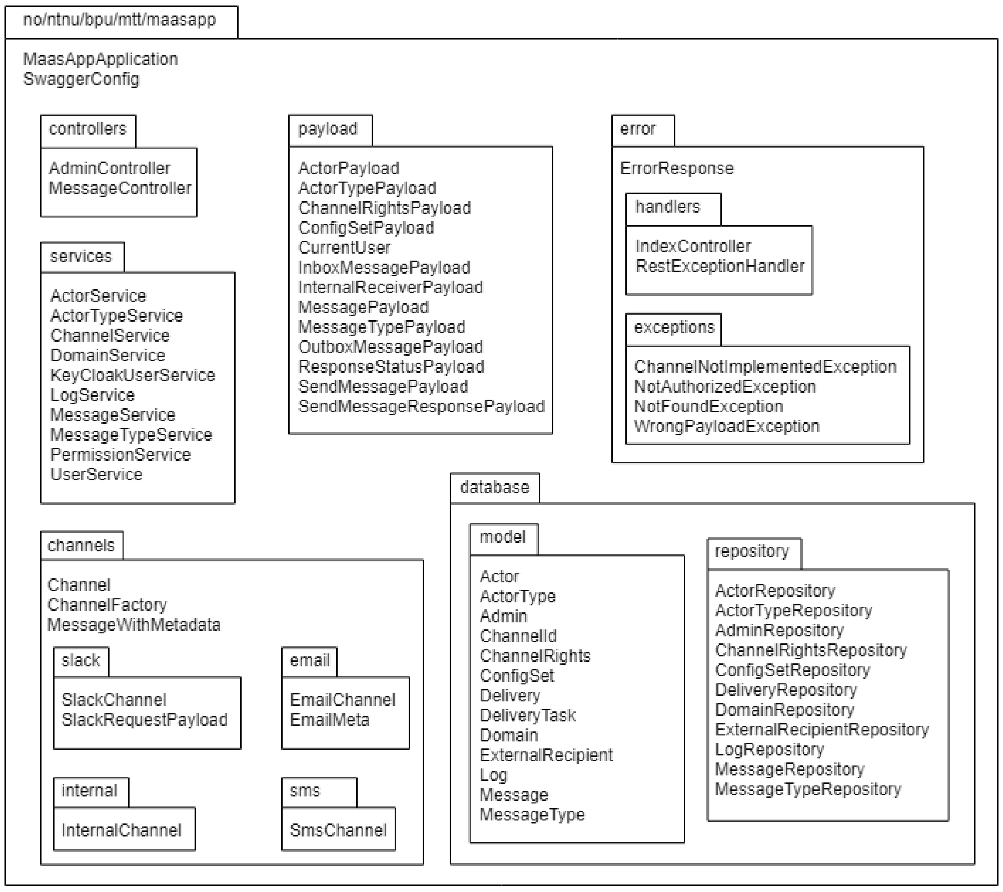


Figure 16: Packages and classes organisation on backend

5.2.2 Frontend

Files and components organisation on frontend is shown on fig. 17. The main “app” component and all the others placed in “components” folder are marked with a light blue colour (files they consist of are omitted here). All the files in “classes” folder define objects that store data on the frontend side. One additional module responsible for routing is placed in the “module” folder. A function used to initialise the application is in “utils”. The implementations of all services are in “services” folder.

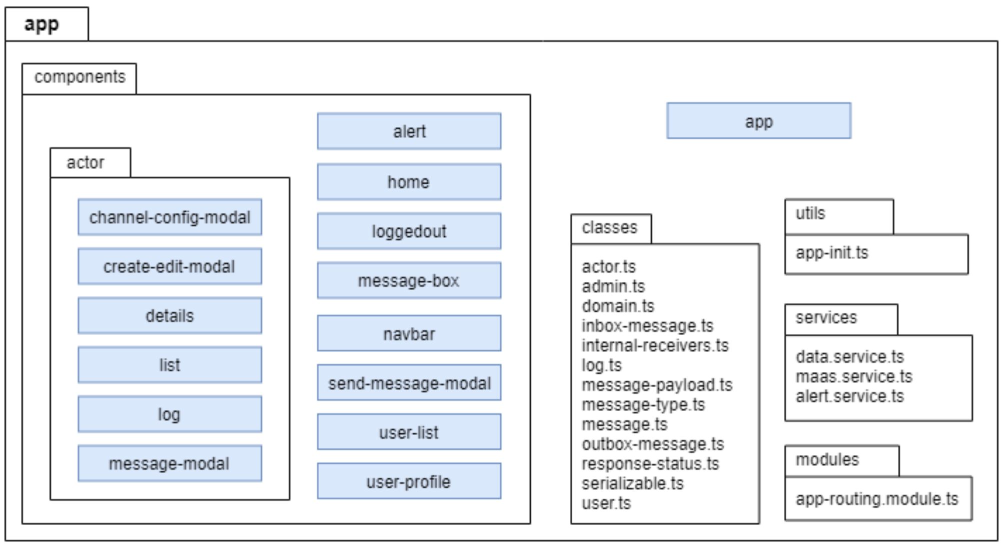
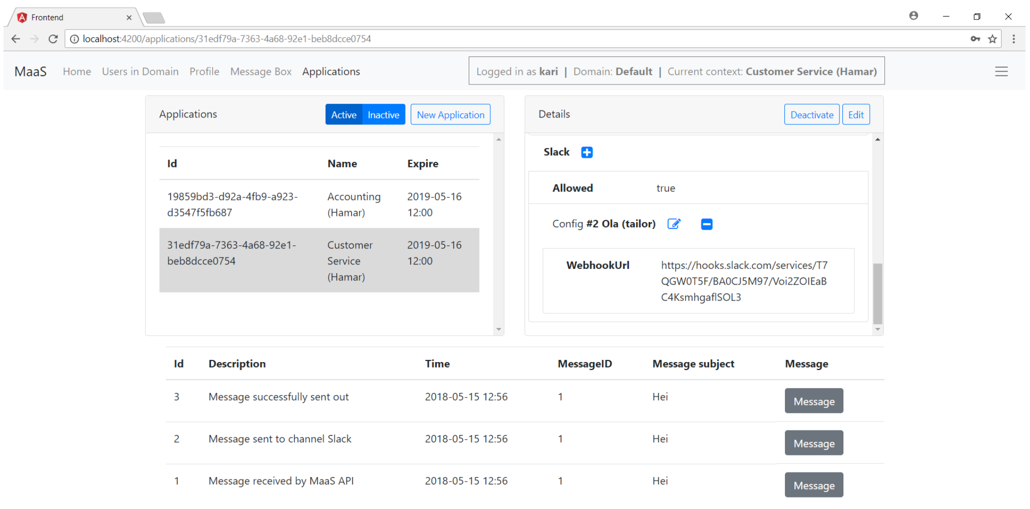


Figure 17: Files and components organisation on frontend

5.3 Application Management - Admin Panel

The MaaS application has a web interface that can be used to:

- Add/Administrate/Delete Applications
- Send/Read messages or check delivery status (log).



Logged in as **kari** | Domain: **Default** | Current context: **Customer Service (Hamar)**

Applications

Id	Name	Expire
19859bd3-d92a-4fb9-a923-d3547f5fb687	Accounting (Hamar)	2019-05-16 12:00
31edf79a-7363-4a68-92e1-beb8dcce0754	Customer Service (Hamar)	2019-05-16 12:00

Details

Slack

Allowed	true

Config **#2 Ola (tailor)**

WebhookUrl	https://hooks.slack.com/services/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

Id	Description	Time	MessageID	Message subject	Message
3	Message successfully sent out	2018-05-15 12:56	1	Hei	Message
2	Message sent to channel Slack	2018-05-15 12:56	1	Hei	Message
1	Message received by MaaS API	2018-05-15 12:56	1	Hei	Message

Figure 18: Screenshot from Admin Panel

The web interface (fig. 18 presents a UI example - an application management view) has been developed using Angular 5 (TypeScript and HTML) and Bootstrap for Angular. The view example includes components (details later in this section):

- navbar - presents navigation links, status (current logged user, his/her assigned domain, “current context” i.e. currently selected App) and “logout” button.
- actor.list - left panel, shows a list (separate lists for active and inactive Apps) of App under control of current user; here also possibility to add a new App
- actor.details - right panel, App details and channels access configuration; on the example view (fig. 18) a configuration of a Slack webhook for the selected App.
- actor.log - bottom panel, shows log entries created during processing of messages send by the selected App (default sorting - latest first); here also a link to a message preview with delivery status for each recipient.

The web application consists of following elements, developed according to Angular documentation [9]:

- **Thirteen classes:** They represents different objects like an application, a message, or a log. This objects can be used by the the components (views) using their methods or injecting some or their data in the views. Objects from these classes can as well be sent to the backend converting them into a JSON payload.

```
export class Actor extends Serializable {
    id: string;
    name: string;
    description: string;
    expire: Date;
    type: ActorType;
    keyAdmins: Object[];
    channelRights: ChannelRights[];
    availableTypes: string[];
...
    setChannelRightsAllowed(channelType:string, allowed: boolean) : boolean {
        let needNewCr = true;
        this.channelRights.forEach( cr => {
            if (cr.messageType == channelType) {
                cr.allowed = allowed;
                needNewCr = false;
            }
        });
        return needNewCr;
    }
...
}
```

Listing 5.2: Extract of Actor class

- **Two Modules:** The main module “app.module.ts” provides a compilation context for all components while the module “app-routing.module.ts” has the responsibility for changing the components that are shown according to the web address that has been requested i.e. if the address “/profile” is introduced, the application will show detail information about currently logged user.

```
const routes: Routes = [
    {path: '', component: HomeComponent},
    {path: 'users', component: UserListComponent},
    {path: 'profile', component: UserProfileComponent},
    ...
    ]
```

Listing 5.3: Extract of the Routing Module

- **Fourteen Components:** A component represents a single element of the application such as the home page, or the view used to show the message box of an application. Every component consist of four files:

1. **(.scss) file:** In Angular it is possible to define individual css style to a single component.
2. **(.spec.ts) file:** This is a unit tests file
3. **TypeScript file (.ts):** This file is the data logic of a component and it is written in TypeScript. The variables declared in this file can be injected in the view (HTML file) using Dependency Injection pattern [9]. A constructor and other methods are defined as well. They can be called and used by the view if necessary. Two methods: ngOnInit() and ngOnDestroy() are executed when a component is opened or closed.

```
export class MessageBoxComponent implements OnInit, OnDestroy {

    inboxMessages: InboxMessage[];
    actors: Actor[];
    currentActor: Actor;

    private currentActorSubscription: ISubscription;
    private actorsSubscription: ISubscription;
    private inboxMessagesSubscription: ISubscription;
    private routeSubscription: ISubscription;

    constructor(
        public dataService: DataService,
        private maasService: MaasService,
        private modalService: NgbModal,
        private route: ActivatedRoute
    ) { }

    ngOnInit() {
        this.routeSubscription = this.route.params
                                .subscribe(params => {
            console.log("route change");
            if(params.id) {
                this.dataService.setActorById(params.id);
            }
        });

        this.currentActorSubscription = this.dataService.currentActor
                                    .subscribe(actor => {
            this.currentActor = actor;
            if(actor != null) {
                console.log("udating current actor inbox");
                this.dataService.updateInboxMessages(actor.id);
            }
        });

        this.getAllInboxMessages();
        this.actorsSubscription = this.dataService.getActors()
                        .subscribe(actors => this.actors = actors);
    }
...
}
```

Listing 5.4: Extract of the .ts file of the Actor component.

4. **HTML file:** This file is the view that will be shown; it is possible to bind some data logic (from a .ts file). An example of this bind is in listing 5.5; message headers shown in the view have different font style and text content depending on inboxMessage object attributes.

```
<ngb-accordion [closeOthers]="true" activeIds="static-1">
  <ngb-panel *ngFor="let inboxMessage of inboxMessages">
    <ng-template ngbPanelTitle>
      <div class="row">

        <div class="col-2 text-left text-info"
        [class.font-weight-bold]="!inboxMessage.delivered">
        <span *ngIf="inboxMessage.delivered; then read else unread"></span>
          {{inboxMessage.subject}}
        </div>

        <div class="col-2 text-left text-dark"
        [class.font-weight-bold]="!inboxMessage.delivered">
          {{inboxMessage.senderName}}
        </div>

        <div class="col-4 text-left text-truncate text-muted"
        [class.font-weight-bold]="!inboxMessage.delivered">
          {{inboxMessage.content}}
        </div>

        <div class="col-4 text-right text-dark"
        [class.font-weight-bold]="!inboxMessage.delivered">
          {{inboxMessage.time | date: 'medium'}}
        </div>

      </div>
    </ng-template>
    <ng-template ngbPanelContent>
      {{inboxMessage.content}}
    </ng-template>
  </ngb-panel>
</ngb-accordion>
```

Listing 5.5: Extract of the Message Box html file

- **Three Services:** Services are used to implement a data logic that are not directly related with an specific component. The data from the services is shared by the components. This data can be e.g. injected in the views of the components or used to carry out some logic operations. In this application DataService class ("data.service.ts" file) is used to share data of the selected current actor between all the components. MaasService class ("Maas.service.ts" file) is used to make the necessary API requests to the backend. In listing 5.4 it is shown how the MessageBox component uses the DataService to fetch a message inbox of a Current Actor. In listing 5.6 it is shown how the MaasService creates a HTTP GET Request to the Backend API and expects response as an array of InboxMessage objects as JSON (httpOptions contains HTTP Header information) in order to get the message inbox of a certain actor. In case of an error response from API the third one, AlertService is responsible to present the error message to the user.

```
@Injectable()
export class MaasService {
    const httpOptions = {
        headers: new HttpHeaders({
            'Content-Type':  'application/json',
        })
    };
...
    getInboxMessages(actorId:string):Observable<InboxMessage[]>{
        return this.http.get<InboxMessage[]>(
            this.serviceURL + "/db/actors/"+actorId+"/inbox", httpOptions
        ).catch(err => this.errorHandler(err));;
    }
...
}
```

Listing 5.6: Extract of Maas Service

5.3.1 Login

The Admin Panel is protected by Keycloak. It means that an admin that is not logged, will be redirected to the Keycloak service; this is realised by KeycloakService.init method from the “keycloak-angular” [15] library. The method is called during frontend initialisation (listing 5.7). This library has an HttpClient Interceptor that adds an authorisation header to the HttpClient requests. If an user introduces a right username and password combination an authorisation token from Keycloak will be received. This token will identify the user as an authenticated and authorised user in the Admin Panel. Once a user is authorised by Keycloak it will not be necessary to make a new login as long as the token is valid (token expiration time can be adjusted in Keycloak Admin Console) or in case the user logs out of the Admin Panel. The token that identifies the user is stored in the web browser.

```
...
    await keycloak.init({
        initOptions: {
            onLoad: 'login-required',
            checkLoginIframe: false
         }
    });
...
```

Listing 5.7: Extract of App-init.ts. Integration of Keycloak and Angular.

5.3.2 Add Application

An App Admin can register a new application (App) in MaaS through the web interface. The component “actor-create-edit-modal” will display a modal where it is necessary to introduce some information like the name of the application, how long the application will have access to MaaS and which channels the application can use. If all the necessary information is validated (see listing 5.8), the component will make a new Actor object and it will call to the MaasService (see listing 5.9). This service will invoke the necessary REST API call to create and save a new actor. Due this API call is protected by Keycloak, the service will send a HTTP Request that will contain the authorisation token and the new Actor Object as JSON payload.

```
...
 <div class="modal-body">
  <ngb-alert *ngIf="f.invalid" [dismissible]="false" type="danger">
    <strong>Error</strong> Missing input!
  </ngb-alert>
  <form #f="ngForm" (ngSubmit)="onSubmit(f)">
    <div class="form-group">
      <label for="inputName">Name</label>
      <input ngModel required name="name" type="text" class="form-control"
      id="inputName" placeholder="Name your application">
    </div>
      ...
  </form>
...
```

Listing 5.8: Extract of the actor-create-edit-modal.component.html file used to create and edit actors.

```

...
  private onUpdate(actor: Actor, next: (actor: Actor) => any) {
    this.maasService.updateActor(actor, this.actor.id).subscribe(next);
  }

...

 private onCreate(actor: Actor, next: (actor: Actor) => any) {
    this.maasService.addActor(actor).subscribe(next);
  }

...
```

Listing 5.9: Extract of the actor-create-edit-modal.component.ts file used to create and edit actors.

5.3.3 Add Configuration

Once App Admin has created an App, it is possible to add configurations to it. A configuration is a set of instructions that contains predefined, additional information about how a message should be sent e.g. using a certain sender address, message subject or receiver email. Configurations can be added, edited and removed. They consist of a set of properties defined for every kind of implemented channel. The "actor-details" component shows information about a certain application, its channel access rights and the configurations associated for each available channel.

When adding a new configuration, the "channel-config-modal" component will display a modal that will require the necessary properties to create a new configuration for the specific channel. When the necessary data is validated, the component will create a ConfigSet object that will contain the predefined properties and theirs values. So it will call the MaasService function to send a HTTP POST Request to the backend. This request contains an Actor object, with adjusted ChannelRight and ConfigSet, parsed to JSON.

5.3.4 Error Handling

When sending a HTTP Request to the backend (in MaasService), we catch all errors and send them to an error handler function (See listing 5.6).

```
private sendToast(alert: Alert) {
    switch (alert.type) {
        case AlertType.Error:
            this.toastr.error(alert.message, AlertType[alert.type], {disableTimeOut: true});
            break;
        case AlertType.Info:
            this.toastr.info(alert.message, AlertType[alert.type], {disableTimeOut: true});
            break;
        case AlertType.Success:
            this.toastr.success(alert.message, AlertType[alert.type], {disableTimeOut: true});
            break;
        case AlertType.Warning:
            this.toastr.warning(alert.message, AlertType[alert.type], {disableTimeOut: true});
            break;
    }
}
```

Listing 5.10: Extract from alert.component.ts for displaying toasts.

The "errorHandler" function invokes the AlertService which adds this error to a Subject[1]. A Subject is an Observable that a component can subscribe to. The AlertComponent subscribes to this subject and displays a "toast" when it receives an alert. For sending

[1]http://reactivex.io/rxjs/class/es6/Subject.js Subject.html

“toasts” we use ngx-toastr[2] (see listing 5.10 and fig. 19).

Toasts are also sent when an user tries to send an incomplete form.

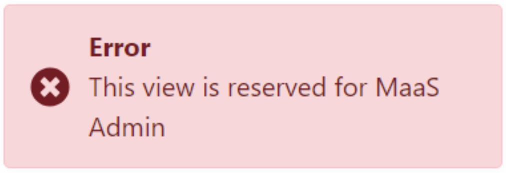

Error
This view is reserved for MaaS Admin

Figure 19: Toast on error in Admin Panel

5.4 Backend API for Application Management

In this subsection we will describe how the process of administrating an Actor is done from backend perspective (CRUD operations). An Actor in this context is an object that represents an application registered in MaaS.

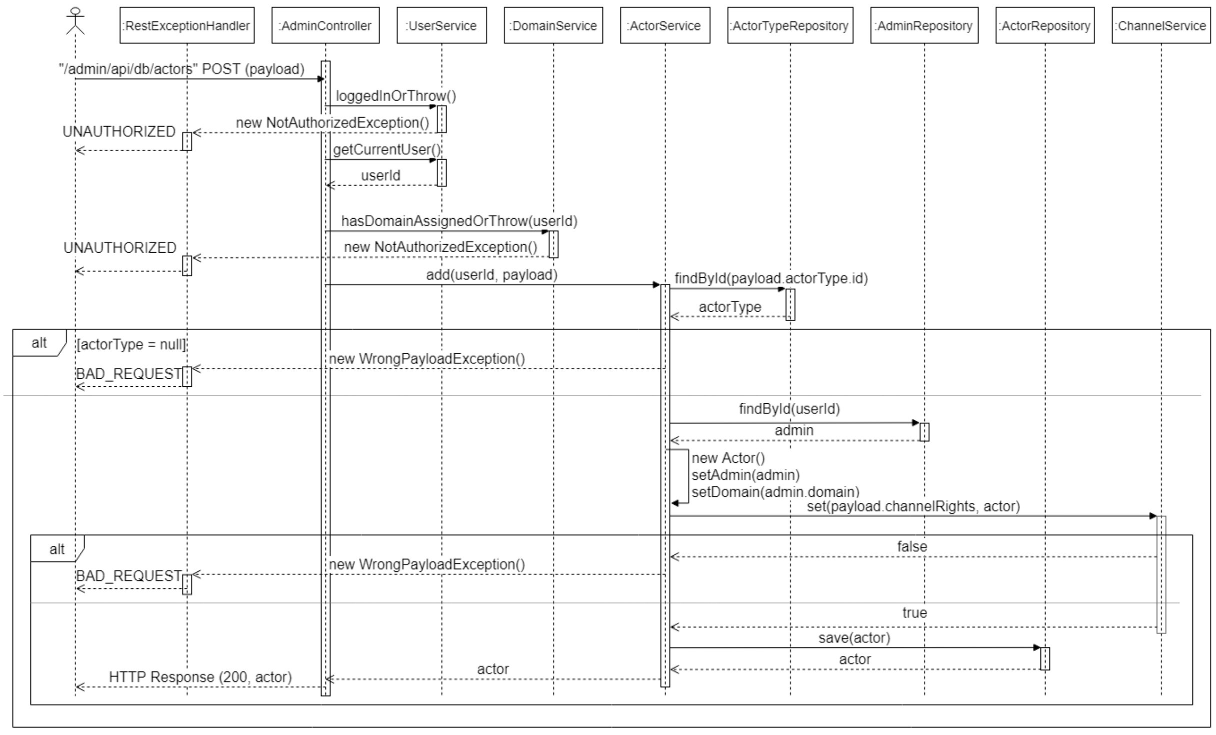


Figure 20: Sequence diagram for adding an Actor

As described earlier in section 3.4, the backend is developed using Spring Framework. The application consists of 6 packages (details later in this section):

- **Controllers:** consist of every controller class handling the requests.
- **Services:** handle the business logic of the application.
- **Database:** all database models and repositories.

[2]https://github.com/scttcper/ngx-toastr

- **Channels:** the Channel Factory and every implemented channel in a separate sub-package.
- **Payload:** POJO classes for representation of JSON objects.
- **Error:** error exceptions and handlers.

A request to the system with the "/admin/api/actors" endpoint is processed by the Admin Controller, and then Actor Service, Actor Repository, Actor Entity; finally data is stored in the database. This process flow (fig. 20) will be explained next.

5.4.1 Admin Controller

Before reaching this controller, the request needs to be authenticated with Keycloak. To achieve this the Spring Boot Adapter for Keycloak[3] has been used. This needed to be configured in the application.properties file by setting the Keycloak URL, realm, resource, and which endpoint to secure ("/admin/api").

The AdminController class is annotated with @RestController and @RequestMapping("/admin/api") to handle all request from this endpoint with this class.

This controller (listing 5.11) is using several Services by Dependency Injection in the class constructor.

```java
@RestController
@RequestMapping("/admin/api")
@Api(value = "Admin Panel Service", description = "Description Admin Panel Service")
public class AdminController {
...
    @Autowired
    AdminController(
            UserService userService,
            ActorService actorService,
            ActorTypeService actorTypeService,
            LogService logService,
            MessageService messageService,
            MessageTypeService messageTypeService,
            ChannelService channelService,
            PermissionService permissionService) {
        this.userService = userService;
        this.actorService = actorService;
        this.actorTypeService = actorTypeService;
        this.logService = logService;
        this.messageService = messageService;
        this.messageTypeService = messageTypeService;
        this.channelService = channelService;
        this.permissionService = permissionService;
    }
...
}
```

Listing 5.11: A constructor from AdminController class

Each function in this class is also annotated with @RequestMapping or similar annotation to narrow down the scope of requests the function should handle. For example for adding a new Actor to the database:

```java
@PostMapping("/db/actors")
@ApiOperation(value = "An admin adds a new actor")
public Actor addNewActor(@RequestBody ActorPayload payload) {
    this.userService.loggedInOrThrow();
    return actorService.add(this.userService.getCurrentUser().getUserId(), payload);
}
```

Listing 5.12: Adding Actor from Admin Controller

[3]https://www.keycloak.org/docs/3.2/securing_apps/topics/oidc/java/spring-boot-adapter.html

Listing 5.12 shows how the function handles requests on the "/admin/api/db/actors" endpoint (narrowed down from "/admin/api'). It is also using the @RequestBody[4] annotation on the payload parameter, by doing this the received JSON request body is bound to this parameter and automatically marshalled to an ActorPayload object by Spring, which is using the Jackson library[5]. This ActorPayload object is a POJO with attributes representing the JSON object; name, description, expire, type and channel rights. After this, the business logic (Actor Service explained in the next section) process the request.

Similar to "addNewActor" shown in listing 5.12, every function in the AdminController class is using the "loggedInOrThrow" function in the UserService class, which checks if the user is logged in, if not, it throws an exception. Error handling is explained later in section 5.4.6.

5.4.2 Actor Service

The Actor Service is a service class as explained in section 3.4.2 which takes care of business logic for CRUD operations on an Actor.

This service, and all other services are using Dependency Injection, to autowire database repositories (explained later in section 5.4.4) and other services, similar to how it's done in Admin Controller (listing 5.11).

Listing 5.13 presents an example of how a new actor is created and added.

```java
public Actor add(String userId, ActorPayload payload) {
    ActorType actorType = actorTypeRepository.findById(payload.getType().getId());
    if (actorType == null) {
        throw new WrongPayloadException("You didn't specify the kind of Actor");
    }

    Actor actor = new Actor();
    actor.setDescription(payload.getDescription());
    actor.setExpire(payload.getExpire());
    actor.setType(actorType);
    actor.setName(payload.getName());

    Admin admin = adminRepository.findById(userId);
    actor.setDomain(admin.getDomain());
    actor.addAdmin(admin);

    if (!channelService.set(payload.getChannelRights(), actor)) {
        throw new WrongPayloadException("You tried to give access rights to" +
            + " a kind of channel that doesn't exist");
    }

    return actorRepository.save(actor);
}
```

Listing 5.13: Add new Actor function

In this function an Actor object which is an Entity class (more details in section 5.4.3) is created based on the ActorPayload. Channel Service sets the correct channel rights, this determines if the actor has access to the specific channel (Email, Internal etc.)

Then the ActorRepository (more in section 5.4.4) is used to save the actor to the database.

[4]https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestBody.html
[5]https://github.com/FasterXML/jackson

5.4.3 Actor Entity

Classes annotated with @Entity are the abstract representations of database tables as explained in section 3.2.3, all these entities are placed in the "database.model" package.

The Actor entity contains several variables that represents columns in a table, for example the "id" column (listing 5.14)

```java
@Id
@GeneratedValue(generator = "uuid")
@GenericGenerator(name = "uuid", strategy = "uuid2")
@Column(
        name = "id",
        length = 36
)
private String id;
```

Listing 5.14: Actor Entity "id" definition

Here the @Id annotation defines this property as an identifier for the entity. This is the primary key in the database. The identifier will be auto-generated using a generator defined by @GeneratedValue and @GenericGenerator, which in this case is a UUID. This unique identifier is used by MaaS in the App authorisation process as an API Key (choice discussion later in section 8.1.2). @Column annotation allows to define different properties, e.g. name (here a column name set to "id"), length (here maximum size of "id" String), unique and nullable constraints [16].

For relations with other entities we use @ManyToMany, @OneToMany and @ManyToOne annotations. For example the set of admins in the Actor entity is defined with a @ManyToMany annotation, because an Actor can have many admins, and an admin can control many Actors.

```java
@ManyToMany (
        cascade = CascadeType.ALL
)
@JoinTable(
    name = "actors_admins",
    joinColumns = { @JoinColumn(name = "actor_id") },
    inverseJoinColumns = { @JoinColumn(name = "admin_id") }
)
private Set<Admin> admins = new HashSet<>();
```

Listing 5.15: Actor Entity "admins" definition

```java
@ManyToMany(
    cascade = CascadeType.ALL,
        mappedBy = "admins"
    )
private Set<Actor> actors = new HashSet<>();
```

Listing 5.16: Admin Entity relation with Actor Entity

This creates a Many-To-Many relation between the Actor and Admin entities. With cascading we can make sure that any operation on the "admins" property from the Actor entity is also applied to the Admin entity [16]. Since this is a Many-To-Many relation, a table, which is defined here by the @JoinTable annotation, it is created automatically. The table is named "actors_admin" and have columns "actor_id" and "admin_id" which represent the Actor and Admin identifiers.

Actor entity can also be marshalled to JSON and returned from the API, but not all information about this object should be returned, to prevent this we use the @JsonIgnore[6] annotation, this excludes the "getter" function from marshalling and this "ignored" Actor attribute will not be returned.

For example we use this in Actor class to prevent the Domain object to be returned to the user. Instead we only want the name of the domain returned so we created a function that return String instead (listing 5.17); the JSON object created based on the Actor object will contain an attribute "domainName" which allow to avoid a problem with a recurrent object. The problem could appear in this case because an Actor object contains a Domain object which contains Actor objects and so on.

```java
...
    @JsonIgnore
    public Domain getDomain() {
        return this.domain;
    }

    public String getDomainName() {
        return domain.getName();
    }
...
```

Listing 5.17: Actor get domain with ignore

5.4.4 Actor Repository

To interact with the database (send queries) we are using Spring Data Repositories[7]. This allows to create an interface without an implementation of e.g. queries (Spring is creating the implementation automatically).

```java
public interface ActorRepository extends CrudRepository<Actor, String> {
    Actor findById(String id);
    Set<Actor> findAllByAdmins_Id(String adminId);
    ...
}
```

Listing 5.18: ActorRepository interface and an example of query method definitions

This interface is a CrudRepository[8] which means it has some pre-defined methods for performing CRUD operations. The CrudRepository takes two type parameters, the Entity class and id type (type of the primary key in the entity). Complete interface in appendix D (listing D.5).

For a more complex "query method" definition we have an example from ConfigSetRepository (listing 5.19). The name of method determines the query to find a ConfigSet based on "configId" and "actorId". This looks in the ConfigSet table, finds the "configId" and looks for the ChannelRights table which has a ChannelId (@EmbeddedId that represents relation between an Actor and a ChannelType) that is linked to an Actor with given "actorId".

```java
ConfigSet findByIdAndChannelRights_ChannelId_Actor_Id(long configId, String actorId);
```

Listing 5.19: ConfigSetRepository ConfigSet by id and Actor

[6]https://fasterxml.github.io/jackson-annotations/javadoc/2.5/com/fasterxml/jackson/annotation/JsonIgnore.html
[7]https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html
[8]https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

We used method names for creating queries in this interface, but it could also be done with the @Query annotation for defining more complex queries; @Query annotation example later in section 5.5.3 while presenting Actor verification during message sending.

5.4.5 Database Representation

All classes annotated with @Entity and relations between them are represented with the structure of tables presented on fig. 21.

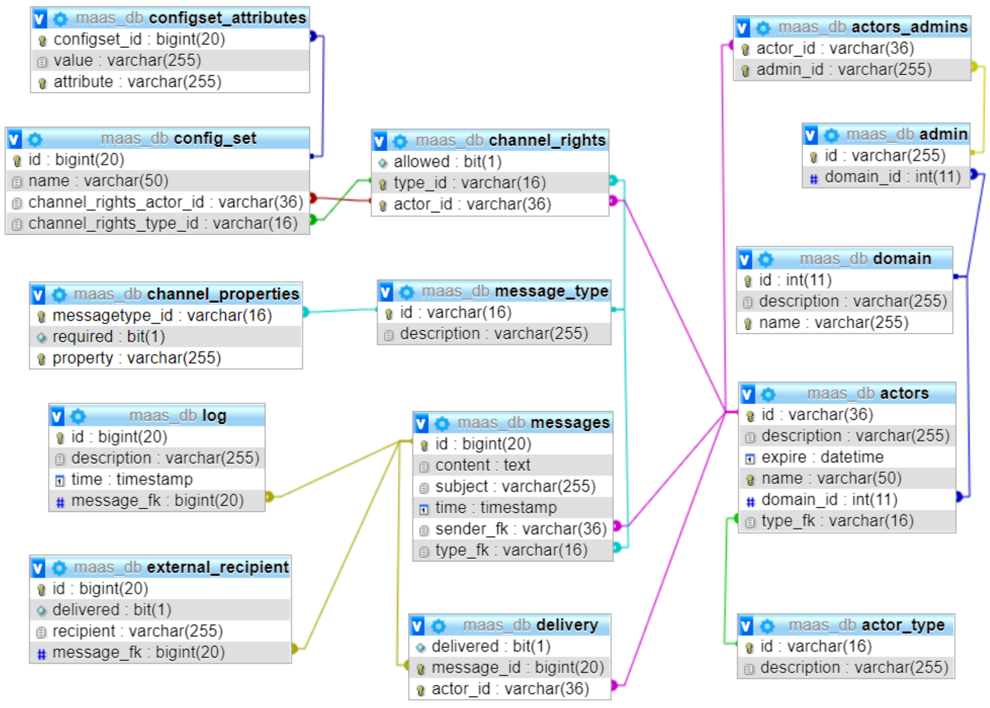


Figure 21: Database structure, phpMyAdmin

5.4.6 Error handling

All Exceptions thrown are handled by the RestExceptionHandler in the “error.handlers” package. Other exceptions are handled by IndexController which overrides the “/error” mapping by Spring to show a custom error message.

The RestExceptionHandler class is annotated with @RestControllerAdvice[9], so that Spring can find the methods in this class which are annotated with @ExceptionHandler[10]. These methods are called when a certain exception is thrown.

```
@ExceptionHandler
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
public ErrorResponse handleWrongPayloadException(WrongPayloadException e) {
    return new ErrorResponse(HttpStatus.BAD_REQUEST , e);
}
```

Listing 5.20: Handler for wrong payload exceptions

[9]https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestControllerAdvice.html

[10]https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html

This function (listing 5.20) runs when a WrongPayloadException is thrown. It returns an ErrorResponse object which is the response to the user (in JSON format). The ErrorResponse object is returned for every exception. This way we ensure that all errors get the same JSON structure. The ErrorResponse object contains a message including the error message.

These functions are also annotated with @ResponseStatus[11]. This will alter the HTTP response to use the appropriate status code. For example “HttpStatus.BAD_REQUEST” will result in a 400.

5.4.7 Documentation

All methods are documented using JavaDoc[12]. In addition an OpenAPI documentation (created with Swagger[13] framework) is created for the API.

To help to automate this process a Springfox[14] library is used. This Java library offers a support for Spring. With use of annotations it is possible to document the API.

Every method in the controllers are annotated with @ApiOperation as seen in listing 5.12 (page 36). As a result, all annotated methods are added to the documentation.

An export of the documentation to a PDF format is attached to this report in appendix E.

5.5 Message Flow for Internal Messages

In this section we will present how a message is processed through the MaaS system. An internal message will be used here as an example. The sequence diagram (fig. 22) presents the first part of message processing, i.e. from receiving HTTP Request to accepting a message for sending (as an alternative an error response is created for messages not accepted for processing).

5.5.1 Application

Here we assume, that in order to be able to send a message, an application has been prepared to interact with MaaS; following steps are necessary:

- An API access key (application id as described previously in section 5.4.3) that has been stored in the application.
- JSON payload is formatted accordingly to API specification (documentation in appendix E).
- The application is connected to internet and able to send HTTP POST Requests (with prepared JSON payload).

[11]https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseStatus.html
[12]http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[13]https://swagger.io
[14]http://springfox.github.io/springfox/

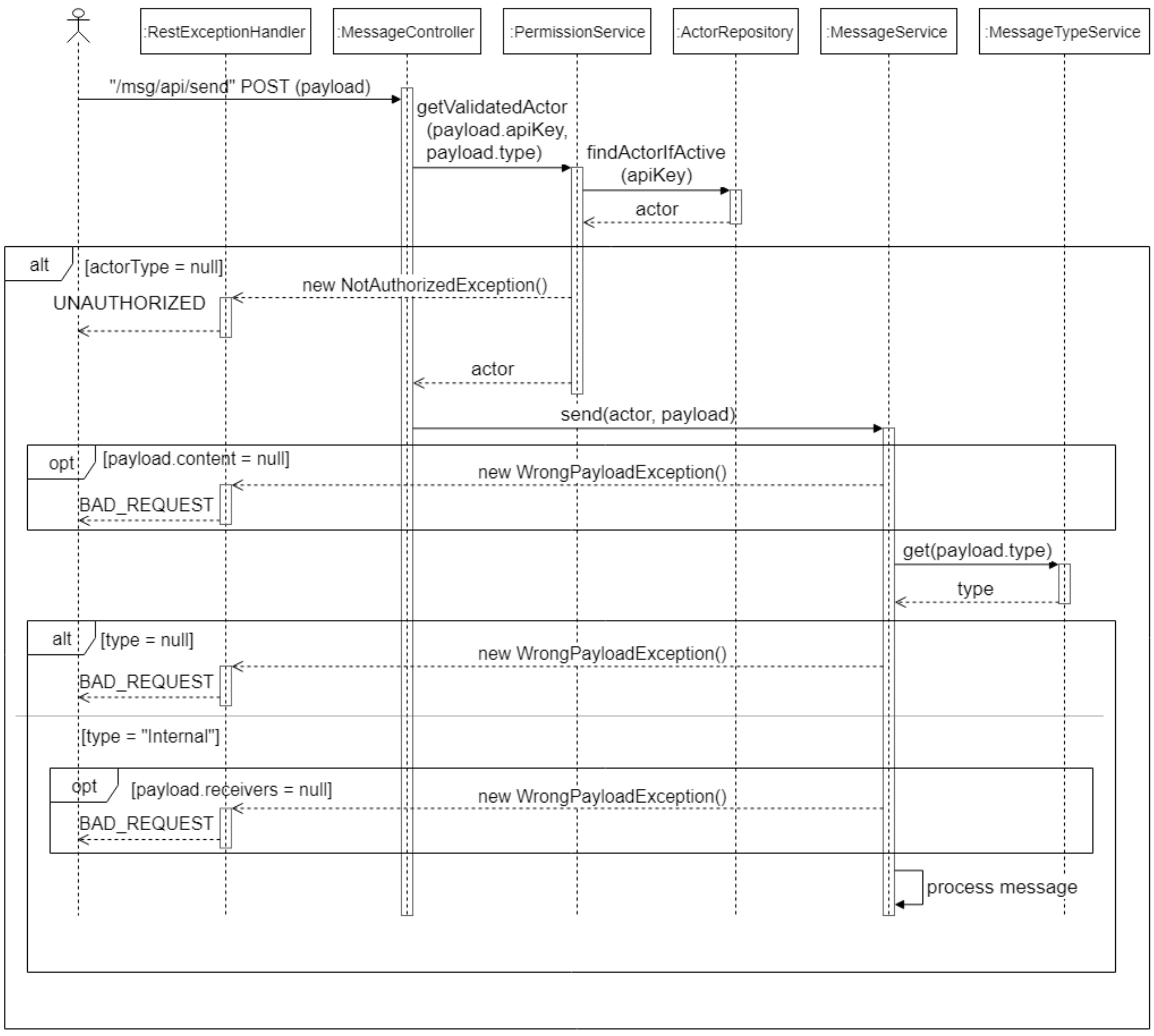


Figure 22: Sequence diagram for message sending, first steps

5.5.2 Request Handling

The next step concerns MaaS. A HTTP Request is handled by a MessageController; which is implemented similar to the AdminController described previously. Listing 5.21 presents the function that handles a request; classes that are involved in parsing a JSON request body are presented below it (listing 5.22, listing 5.23); constructors, getters and setters omitted in listings from those classes.

```
@PostMapping(value = "/send", consumes = "application/json")
@ApiOperation(value = "Sends a message using a certain actor",
        notes = "It is necessary to know the Actor Key" )
public SendMessageResponsePayload sendMessage(@RequestBody OutboxMessagePayload payload){
    Actor actor = permissionService
        .getValidatedActor(payload.getApiKey(), payload.getType());
    return this.messageService.send(actor, payload);
}
```

Listing 5.21: Sending a message from MessageController

```java
public class MessagePayload {
    private String type;
    private List<String> receivers;
    private List<String> recipients;
    private String content;
    private String subject;
    private Timestamp time;
    private Object meta;
    private Long configId;
...
}
```

Listing 5.22: MessagePayload class, header and fields

```java
public class OutboxMessagePayload extends MessagePayload {
    private String apiKey;
    private Set<Delivery> deliveryStatus;
...
}
```

Listing 5.23: OutboxMessagePayload, header and fields

5.5.3 Application Verification

There are some differences, specific to MessageController functionality. Message API is intentionally not secured with Keycloak (choice discussed later in section 8.1.2).

Access Key and Channel Verification

We implemented an own verification system based on API Access Keys (App id) generated in the Admin Panel (as described previously in section 5.4.3). The key, that represents a message sender, and a message type (channel) are extracted from the HTTP Request and send to the PermissionService; this service is responsible for checking permissions of users and registered applications.

```java
public Actor getValidatedActor(String apiKey, String type) {
    Actor actor = this.getValidatedActor(apiKey);

    if (!actor.getChannelRights()
            .stream()
            .filter(cr -> cr.getChannelId().getType().getId().equals(type))
            .findAny()
            .orElse(new ChannelRights())
            .isAllowed()) {
        throw new NotAuthorizedException("No access to channel: " + type);
    }

    return actor;
}

public Actor getValidatedActor(String apiKey) {
    Actor actor = actorRepository.findActorIfActive(apiKey);
    if (actor == null) {
        throw new NotAuthorizedException("No access for given API KEY");
    }
    return actor;
}
```

Listing 5.24: Get validated Actor functions from PermissionService class

Listing 5.24 presents two functions (from PermissionService class), that run a verification process as follows:

- The first step is to obtain an active Actor from ActorRepository (listing 5.25), an exception will be thrown if no Actor (identified by the given API Access Key) is found. Intentionally only one common message for exception is implemented to make probing API for inactive but registered API keys harder.

- When an Actor is found its channel rights are controlled. An exception is thrown if the Actor is not allowed to use the given channel.

The function from ActorRepository class presented in (listing 5.25) is an example of a custom query; @Query annotation is used to extend the query creation mechanism from JPA to more advanced queries.

```
@Query("SELECT a FROM Actor a
        WHERE a.id = :id
            AND (a.expire > current_timestamp OR a.expire IS NULL)")
Actor findActorIfActive(@Param("id") String actorId);
```

Listing 5.25: Get an active Actor (using custom defined query) from ActorRepository

Verification Completed

The verification of an Actor can end with two possible results:

- Verification failed (not existing or inactive actor, channel not allowed to use by an active actor). In this case the exception that was thrown is handled by a common, for the whole system, error mechanism described previously in section 5.4.6.
- Positive verification, i.e. the Actor is active and can use the requested channel; in this case the request payload can be passed to MessageService and the HTTP Response is based on results returned from the service.

5.5.4 Message Handling

Messages sent through MaaS are handled by MessageService. The “send” method receives a validated Actor (an application that is allowed to use MaaS) and a HTTP Request body marshalled into an OutboxMessagePayload object. This method is presented as a whole in appendix D (listing D.1).

Two initial checks (listing 5.26) are done before a message is processed:

- Content not empty. Throws a WrongPayloadException if the message payload doesn’t have a “content” field or the field is empty.
- Message type (requested channel) not found. Throws a WrongPayloadException if the requested message type is not defined in the database.

```
public SendMessageResponsePayload send(Actor actor, OutboxMessagePayload payload) {

    String content = payload.getContent();
    if (content == null || content.isEmpty()) {
        throw new WrongPayloadException("Empty message content is not allowed");
    }

    MessageType type = messageTypeService.get(payload.getType());
    if (type == null) {
        throw new WrongPayloadException("Message type not found");
    }
...
}
```

Listing 5.26: Extract from “send” function from MessageService class

Message Object

The second part of processing a message (presented on a sequence diagram fig. 23) involves ChannelFactory and a specific Channel object for message forwarding.

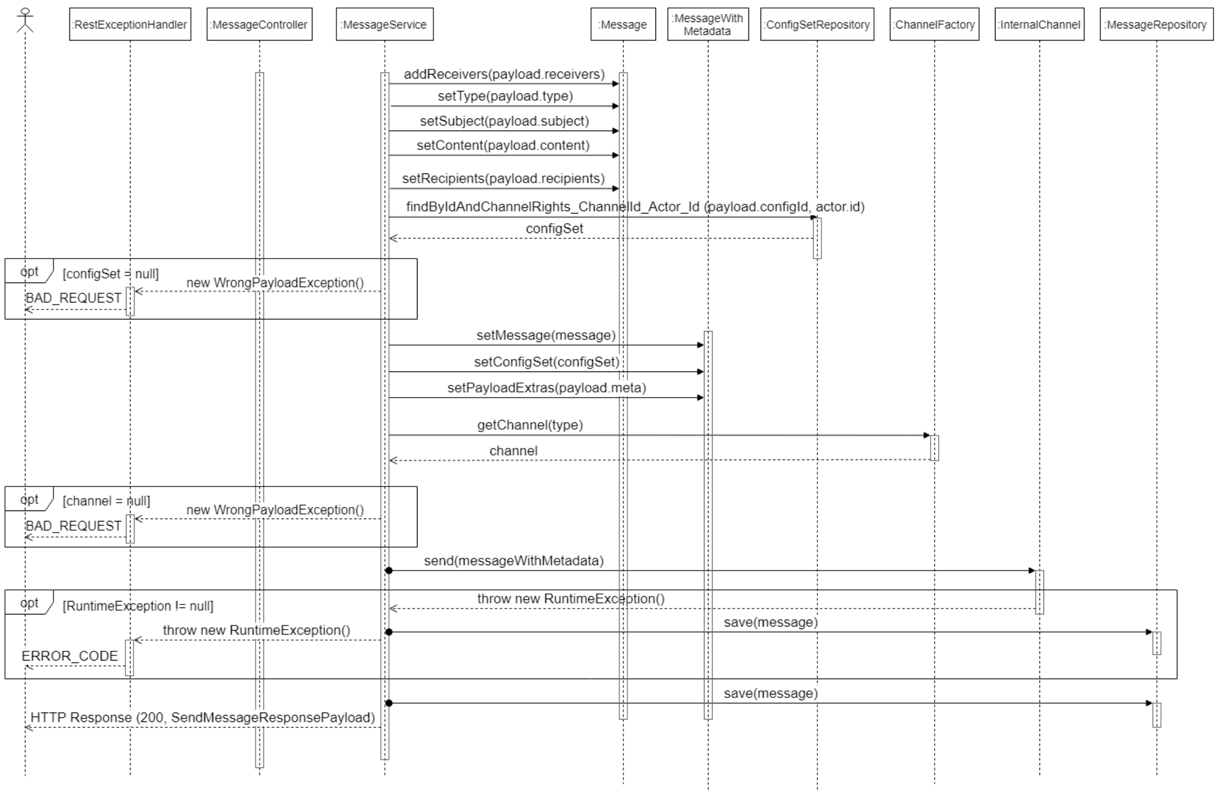

:RestExceptionHandler
:MessageController
:MessageService
:Message
:MessageWith Metadata
:ConfigSetRepository
:ChannelFactory
:InternalChannel
:MessageRepository
addReceivers(payload.receivers)
setType(payload.type)
setSubject(payload.subject)
setContent(payload.content)
setRecipients(payload.recipients)
findByIdAndChannelRights_ChannelId_Actor_Id (payload.configId, actor.id)
configSet
opt
[configSet = null]
new WrongPayloadException()
BAD_REQUEST
setMessage(message)
setConfigSet(configSet)
setPayloadExtras(payload.meta)
getChannel(type)
channel
opt
[channel = null]
new WrongPayloadException()
BAD_REQUEST
send(messageWithMetadata)
opt
[RuntimeException != null]
throw new RuntimeException()
save(message)
throw new RuntimeException()
ERROR_CODE
save(message)
HTTP Response (200, SendMessageResponsePayload)

Figure 23: Sequence diagram for message sending, final steps

When a message passes the initial check it is possible to create a Message object based on the payload (listing 5.27). First, the objects of SendMessageResponsePayload (details about a HTTP Response to the message sender in section 5.5.5) and Message classes are created. In case of internal messages, a valid receiver is an Actor which is in the same Domain as the sender (cross-domain messages are not allowed), has access to “Internal” channel and is active (not expired access rights). A “getReceiver” function finds (in the request payload) all valid receivers based on the sender domain and an array of Actor names (intentionally used unique names instead of id which could allow sending a message using an API Key of another App); later it updates the response object with a status for each requested receiver (see listing D.2 in appendix D); if the set of Actor objects returned from the function is not empty, it means that at least one receiver is valid. In this case the set of Actor object and additional data are bound to the Message object:

- Channel type (MessageType object).
- Message content (String).
- Message subject (String).

- Sender (Actor object).
- External recipients (set of strings, optional if requested to identify App Users i.e. additional information that can be interpreted by App only, MaaS only forwards it along with a message)

```
...
SendMessageResponsePayload customResponse = new SendMessageResponsePayload();
Message message = new Message();

if (type.getId().equals("Internal")) {
    Set<Actor> receivers = getReceivers(
                              actor.getDomain().getId(),
                              payload.getReceivers(),
                              customResponse );

    if (receivers.isEmpty()) {
        throw new WrongPayloadException(
            "Empty receivers list is not allowed for type:" + type.getId());
    }

    message.addReceivers(receivers);
    ...
}
...
```

Listing 5.27: Extract from "send" function from MessageService class

Additional Message Metadata

A message sent to MaaS can contain additional information, as e.g. a predefined configuration or channel specific data. A class MessageWithMetadata (listing 5.28) was introduced to handle also custom metadata that may not correspond with the database structure (for Message); the metadata for currently implemented channels (but also in future extensions of available channels) is handled by Channel implementations (explained later in Message Processing section).

```
public class MessageWithMetadata {
    private Message message;
    private Object payloadExtras;
    private ConfigSet configSet;
    ...
}
```

Listing 5.28: Extract from MessageWithMetadata class, consrtuctor, getters and setters omitted

The "getMetadata" (listing D.3 in appendix D) function creates a MessageWithMetadata object based on the request payload and a configuration found in the database; so the Message object is bound to it. The function "setDefaultDataFromMetadata" (listing D.4 in appendix D) is used to set basic message information like e.g. a subject, if not defined in the Message object itself, but found in the fetched ConfigSet.

```
...
MessageWithMetadata messageWithMetadata = getMetadata(
            payload.getConfigId(), payload.getMeta(), actor.getId());
    messageWithMetadata.setMessage(message);
    setDefaultDataFromMetadata(messageWithMetadata);

    message.addLog(new Log(message, "Message received by MaaS API"));

    sendMessageToProcessing(messageWithMetadata);

    return customResponse;
}
```

Listing 5.29: Extract from "send" function from MessageService class

On this stage the first log entry is created for the accepted message. The Log class (listing 5.30) has an @Entity annotation and represents log data stored in the database related to a message. More log entries examples, concerning different stages of a message processing, in the next section.

```java
@Entity
public class Log {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @ManyToOne
    @JoinColumn(name = "message_fk")
    private Message message;

    private String description;

    @Column(
        columnDefinition = "TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP"
    )
    private Timestamp time;
...
}
```

Listing 5.30: Extract from Log class, constructors, getters and setters omitted

Message Processing

A “sendMessageToProcessing” function (listing 5.31) from MessageService class gets first a concrete Channel object from ChannelFactory (listing 5.32).

```java
private void sendMessageToProcessing(MessageWithMetadata messageWithMetadata) {
    Message message = messageWithMetadata.getMessage();
    String channelType = message.getType().getId();

    Channel channel = this.channelFactory.getChannel(channelType);
    if (channel == null) {
        throw new WrongPayloadException("Channel not found for type " + channelType);
    }

    message.addLog(new Log(message, "Message sent to channel " + channelType));
    try {
        channel.send(messageWithMetadata);
        message.addLog(new Log(message, "Message successfully sent out"));
    } catch (RuntimeException e) {
        message.addLog(new Log(message, "Error sending message: " + e.getMessage()));
        messageRepository.save(message);
        throw e;
    }
    messageRepository.save(message);
}
```

Listing 5.31: Function “sendMessageToProcessing” from MessageService class

```java
public class ChannelFactory {
    public Channel getChannel(String type) {
        switch (type) {
            ...
            case "Internal": return new InternalChannel();
            default:         return null;
        }
    }
}
```

Listing 5.32: ChannelFactory class

In the case of sending an internal message there is no need for a complex implementation of Channel interface (listing 5.33), that is why an InternalChannel class (listing 5.34) can implement "send" as an empty function, i.e. no additional processing is required. External messages need to contact external services, that will take over sending after a message leaves MaaS, so "send" function implementation differs for each channel.

```java
public interface Channel {
    void send(MessageWithMetadata message);
}
```

Listing 5.33: Channel interface

```java
public class InternalChannel implements Channel {
    @Override
    public void send(MessageWithMetadata message) { }
}
```

Listing 5.34: InternalChannel class

If the requested Channel is not found a WrongPayloadException will be thrown. Otherwise a new log entry, informing that a message was sent to a channel, is created and a MessageWithMetadata object is sent to the correct Channel by calling its "send" method. The next log entry depends on results of the message processing by a "send" function which will throw en exception if something went wrong. Since different channel may implement throwing of different exceptions, they are caught as a general RuntimeException and thrown forward to exception handler (as described in section 5.4.6); to store both message and logs in database, also in case of errors during sending, a message need to be saved using MessageRepository.

5.5.5 Response to Application

If no exception has been thrown until "sendMessageToProcessing" has returned, the HTTP Response is generated based on the SendMessageResponsePayload (listing 5.35) object created (by "send" function from MessageService) during the whole process of sending a message. As mentioned before (in section 5.4.6) exceptions are handled by RestExceptionHandler.

```java
public class SendMessageResponsePayload {
    private List<ReceiverDetails> receivers;
    private String message;
    private int code;

    ...

    public static class ReceiverDetails {
        private int code;
        private String actor;
        private String message;
    ...
    }
}
```

Listing 5.35: SendMessageResponsePayload and inner class ReceiverDetails, headers and fields

5.6 Additional Steps for External Messages

The initial steps in a message processing are similar to those described in section 5.5, but the differences, concerning external messages, are presented in this section on a webhook example. The chosen provider of the webhook is Slack[15]. When the message has reached the Slack webhook it is available for a Recipient e.g. in a Slack desktop application; screenshot from the application presented later in this section.

5.6.1 External Message Handling

The “receivers” attribute from a request payload (representing internal receivers of the message) is expected to be “null”; on the other hand “recipients” attribute is the one that contains information about where to send the message, i.e. webhook URLs (in context of different channels it could be e.g. a set of emails, phone numbers etc.). The creation of a Message object, the fetching of additional metadata and the Channel object creation are processed as for internal messages.

5.6.2 Message Converting

The external channel implementation must extract data, that is necessary for converting, from a MessageWithMetadata object. In SlackChannel class (listing 5.36) example, a “send” function expects, that the list of valid URLs is available.

```java
public class SlackChannel implements Channel {
    @Override
    public void send(MessageWithMetadata message) {
        List<URL> urls = getUrls(message);

        if (urls.size() == 0) {
            throw new WrongPayloadException("No valid url");
        }

        for (URL url : urls) {
            sendToSlack(url, message.getMessage());
        }
    }
    ...
}
```

Listing 5.36: Extract from SlackChannel class

```java
private List<URL> getUrls(MessageWithMetadata message) {
    Set<ExternalRecipient> toSet = message.getMessage().getRecipients();
    List<URL> urls = new ArrayList<>();
    for (ExternalRecipient to : toSet) {
        try {
            urls.add(new URL(to.getRecipient()));
        } catch (MalformedURLException e) { }
    }

    ConfigSet configSet = message.getConfigSet();
    if (configSet != null) {
        String url = configSet.getAttributes().get(ATTR_URL);
        if (url != null) {
            try {
                urls.add(new URL(url));
            } catch (MalformedURLException e) {
                throw new WrongPayloadException("Wrong url format in config: " + url);
            }
        }
    }
    return urls;
}
```

Listing 5.37: Function “getUrls” from SlackChannel class

[15]https://api.slack.com/incoming-webhooks

Data is extracted by a “getUrls” function (listing 5.37). First, for each recipient found in Message object, a new URL object is created. The first MalformedURLException is ignored, since it is possible that one of the other strings (from recipients list) could be converted into a valid URL, and the “send” function would throw WrongPayloadException, indicating missing valid URL, if none of the given recipients is approved. Here we apply the same policy as with an internal message, i.e. send forward if at least one recipient is valid (issue discussed during the Review Meeting after the fourth sprint, see appendix J.9.1). Different situation concerns the second MalformedURLException in case of fetching a webhook URL from a predefined configuration, here if an App (sender) uses a ConfigSet prepared by its App Admin and of some reason this configuration is not valid, the message will not be sent but saved in the database with an error log (log view available in e.g. admin panel) to allow future troubleshooting of App configuration.

When all the webhook URLs have been created, each of them is sent to a “sendToSlack” function (listing 5.38). This function will prepare a HTTP POST Request using packages from Spring framework and send it to a given URL; as in the case of internal messages an eventual exception will be caught by MessageService and and re-thrown after creating a log entry. An URISyntaxException could be caused by an invalid URL format, which has been already checked before, but Java requires to catch an exception here while converting it to URI; the other exception will be thrown if a response from Slack webhook doesn’t have a 200 code (the response with a code 200 means that the webhook exists and the message has a correct format).

```java
private void sendToSlack(URL url, Message message) {
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);

    SlackRequestPayload payload = new SlackRequestPayload(message.getContent());
    payload.setUsername(message.getSender().getName());

    String subject = message.getSubject();
    if (subject != null && !subject.isEmpty()) {
        payload.setSubject(subject);
    }


    HttpEntity<SlackRequestPayload> entity = new HttpEntity<>(payload, headers);


    RestTemplate restTemplate = new RestTemplate();
    try {
        ResponseEntity<String> response = restTemplate.postForEntity(
                url.toURI(), entity, String.class);
        if (response.getStatusCode().value() != HttpStatus.OK.value()) {
            throw new WrongPayloadException(
                "Error response from Slack(" + response.getStatusCode()
                + "): " + response.getBody());
        }
    } catch (URISyntaxException e) {
        throw new WrongPayloadException("Wrong URL format");
    }
}
```

Listing 5.38: Function “sendToSlack” from SlackChannel class

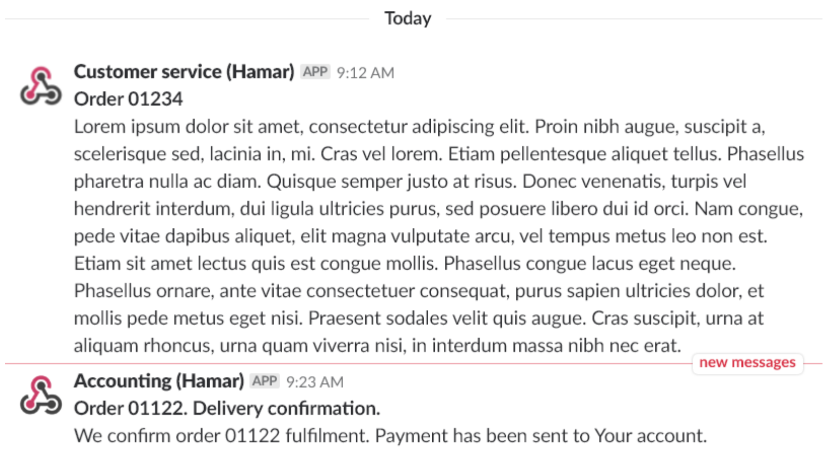
A JSON payload (i.e. fields required by a Slack webhook[16]) is represented by a SlackRequestPayload class (listing 5.39). Message data (“content”, “from”, “subject”) is converted into class attributes (“text”, “username”, “subject”); additionally a “mrkdwn” field indicates that a Markdown is enabled for formatting a message in a receiver interface (Slack desktop application or web-browser).

```java
public class SlackRequestPayload {
    private String text;
    private String username;
    private String subject;
    private boolean mrkdwn;

    public SlackRequestPayload(String text) {
        this.text = text;
        this.mrkdwn = true;
    }
...
}
```

Listing 5.39: SlackRequestPayload class

Below an example of messages sent through MaaS using a Slack webhook channel, as it looks in a receiver interface (fig. 24 is a screenshot from a Slack desktop application connected to a webhook).


Today

Customer service (Hamar) APP 9:12 AM
Order 01234
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin nibh augue, suscipit a, scelerisque sed, lacinia in, mi. Cras vel lorem. Etiam pellentesque aliquet tellus. Phasellus pharetra nulla ac diam. Quisque semper justo at risus. Donec venenatis, turpis vel hendrerit interdum, dui ligula ultricies purus, sed posuere libero dui id orci. Nam congue, pede vitae dapibus aliquet, elit magna vulputate arcu, vel tempus metus leo non est. Etiam sit amet lectus quis est congue mollis. Phasellus congue lacus eget neque. Phasellus ornare, ante vitae consectetuer consequat, purus sapien ultricies dolor, et mollis pede metus eget nisi. Praesent sodales velit quis augue. Cras suscipit, urna at aliquam rhoncus, urna quam viverra nisi, in interdum massa nibh nec erat.

new messages

Accounting (Hamar) APP 9:23 AM
Order 01122. Delivery confirmation.
We confirm order 01122 fulfilment. Payment has been sent to Your account.

Figure 24: Incoming messages from MaaS, screenshot from a Slack desktop application

5.6.3 Response to Application

A response to a message sender is similar as in the case of internal messages. If no exceptions has been thrown along the message processing, then MessageController responds with code 200 and the response payload is a JSON structure marshalled from the SendMessageResponsePayload object.

[16]https://api.slack.com/incoming-webhooks

6 Deployment

In this chapter we will describe how MaaS is prepared for deployment. It hasn't been deployed to an actual production environment, because stakeholders are yet to decide which cloud deployment platform to use; but they wanted the system to be dockerized for easy deployment at a later stage. This will be realised by creating a Docker Compose file explained in this chapter.

The Docker technology has been used during development to run Keycloak, MariaDB and phpMyAdmin services. The phpMyAdmin service is only expected to run in the development environment, since this is not strictly needed in production, although it might be useful for a direct communication with the database without use of API and/or SQL command line.

6.1 Docker

Docker[1] is a software that can create and orchestrate containers. A container is a package that contains everything the application needs to be able to run. These containers are similar to Virtual Machines, but more lightweight because they share the same kernel as the operating system. At the same time they have some of the benefits regarding isolating resources and allocation. A Virtual Machine is an abstraction at the physical layer, a container is an abstraction at the application layer [17].

6.1.1 Docker Toolbox

Docker makes it possible to create and run images (as containers) with the docker command line[2]; to be able to run docker commands on Windows, Docker Toolbox[3] is used because the native docker application for Windows is not compatible with VirtualBox[4] [18]. Docker Toolbox creates a small Linux VM which is used to run docker features that are not able to run native on Windows (without Hyper-V[5]).

6.1.2 Docker Compose

Docker Compose[6] is used to efficient defining and running multiple docker containers by creating a YAML[7] file to configure different services, which can then be built and started with a single command.

This file (usually named "docker-compose.yml") contains a configuration of each service. It is possible to e.g. use a pre-defined image from Docker Hub[8] (which is a repository for docker images) or build from a Dockerfile.

[1]https://www.docker.com
[2]https://docs.docker.com/engine/reference/commandline/cli/
[3]https://docs.docker.com/toolbox/toolbox_install_windows/
[4]https://www.virtualbox.org
[5]https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/
[6]https://docs.docker.com/compose/overview/
[7]http://yaml.org
[8]https://hub.docker.com

6.1.3 Dockerfile

This is a text file that contains information on how to build a Docker image; such as every command needed to build and run the image as a container.

6.2 Reused Docker Images

In this section we will explain how the MariaDB and phpMyAdmin (PMA) images were used in the Docker Compose file.

For MariaDB and PMA we used the existing images from Docker Hub [19, 20], it was only needed to add environment variables and volumes. That is why there was no need to create Dockerfiles for those images.

```
...
    # MariaDB - SQL database
    db:
        image: "mariadb:10.2.12"
        ports:
            - "3306:3306"
        environment:
            - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
            - TZ=${TZ}
        volumes:
            # any .sql or .sh file in the db folder will run
            # when creating this container/volume
            - "./db:/docker-entrypoint-initdb.d"
            - db-data:/var/lib/mysql
    # PHPMyAdmin - for database administration
    pma:
        image: "phpmyadmin/phpmyadmin:4.7.7-1"
        ports:
            - "8081:80"
        environment:
            - PMA_HOST=db
        depends_on:
            - db
...
```

Listing 6.1: Extract from Docker Compose file

In listing 6.1 the “db” and “pma” services are listed where an image from Docker Hub is used.

In the “ports” sections the exported ports are defined; the port is exposed in such a way that the service can be reached from the outside of the container.

The environment variables for MariaDB are set in a separate file (“.env”), which includes all variables needed in the Docker Compose file. This is a “secret” file which contains information that could be different for each instance of the entire MaaS system; this file includes information such as usernames, passwords, URLs etc. The environment variable for PMA (“PMA_HOST” in listing 6.1) uses “db” as a value; this makes use of the internal networking in Compose. Each service defined in the Compose file is (by default) in one network where every container can reach each other by their name (here PMA reaches MariaDB with “db” as a hostname) [21].

In the “volumes” section for “db” (listing 6.1), the folder “./db” is mounted to the “/docker-entrypoint-initdb.d” folder inside the container. This is a special folder in the MariaDB image; when the container is started for the first time it will execute all .sh, .sql and .sql.gz files in this folder alphabetically [19]. We use this feature to initialise the database by creating .sql files containing SQL commands to create the database, the table structure, database users and default data (such as e.g. message types). These .sql files are based on a file, which is auto-generated by PMA; initial data has been added to the file.

```sql
INSERT INTO `message_type` (`id`, `description`) VALUES
('Email', 'Email'),
('Internal', 'Message to registered Actor'),
('Slack', 'Message to Slack webhook'),
('SMS', 'Message to external provider');
```

Listing 6.2: Extract from a file used to import SQL data

6.3 New Docker Images

In this section we will explain how we created Docker images, using Dockerfiles, for Keycloak, backend and frontend; and how they were included in the Docker Compose file. These are Docker images that could be used in a production environment.

6.3.1 Keycloak Dockerfile

Keycloak has its own image on Docker Hub [22], but using this image only, it is not possible to import a previously exported realm. Creating a small Dockerfile based on the existing Keycloak image mitigates this.

```
FROM jboss/keycloak:3.4.2.Final

# Adding the realm-export file. consider using a env variable for this
COPY realm-export.json /opt/jboss/keycloak

ENTRYPOINT [ "/opt/jboss/docker-entrypoint.sh" ]
CMD ["-b", "0.0.0.0", "-Dkeycloak.import=/opt/jboss/keycloak/realm-export.json"]
```

Listing 6.3: Dockerfile.keycloak

Listing 6.3 is the Dockerfile created for Keycloak. The “FROM” statement says which image to base the new image on. Then the “realm.export” file is copied to a location inside the container, this file is an export from a Keycloak instance, containing needed information about the realm (described in section 5.1.1). A “keycloak.import” parameter is added to the command for running the Keycloak instance; in this way the previously exported realm is imported to the new instance. It is possible to add existing users already to this file, or do it later from Keycloak Admin Console.

```yaml
...
    kc:
        build:
            context: .
            dockerfile: Dockerfile.keycloak
        ports:
            - "8082:8080"
        environment:
            - KEYCLOAK_USER=${KEYCLOAK_USER}
            - KEYCLOAK_PASSWORD=${KEYCLOAK_PASSWORD}
            - DB_VENDOR=h2
        volumes:
            - kc-data:/opt/jboss
...
```

Listing 6.4: docker-compose.yml Keycloak

The usage of Dockerfiles in Docker Compose requires a specified build context (location where docker commands are run from) and a Dockerfile. (listing 6.4). Everything else is similar to how it was done for MariaDB and PMA (section 6.2).

6.3.2 Backend

Since the backend is developed as a stand-alone Java application the “openjdk” image from Docker Hub[9] can be used, but using “openjdk” only will clutter the container with unnecessary data from the building process and will be larger than needed.

To address this problem, multi-stage builds [23] are used, there are some other ways, like creating multiple Dockerfiles, but using multi-stage builds is the best practice, according to Docker documentation [24]. With multi-stage builds it is possible to have multiple images in one Dockerfile by using the “FROM” statement more then once. Each of these statements have their own base and files can be copied between them.

```
FROM maven:3.5.3-jdk-8-alpine as builder

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

COPY pom.xml /usr/src/app
COPY src /usr/src/app/src

RUN mvn package -DskipTests

RUN mv /usr/src/app/target/maas-app-*.jar /app.jar

FROM openjdk:8u151-jdk-alpine3.7

EXPOSE 8080

COPY --from=builder /app.jar .

CMD ["/usr/bin/java", "-jar", "app.jar"]
```

Listing 6.5: Dockerfile for backend

Listing 6.5 shows that first the “maven” image[10] is used to build the application. Then the .jar file from the builder image (based on “maven”) are copied into the image (based on “openjdk”).

Usage of this Dockerfile in the Docker Compose file is similar to how Keycloak was added, except that the URL to the GIT repository was used as a context, by doing this, the Compose file is independent from the source code; Docker Compose will clone this repository and use it as a context before running any Docker commands [25].

6.3.3 Frontend

Angular 5 is built using the Angular CLI command “ng build –prod” for production, this creates HTML, JavaScript files etc. needed for the Frontend; a Web Server to serve these files is also needed.

As a Web Server, NGINX[11] was chosen because it was one of the recommended servers by Angular [26]; additionally, according to a Netcraft survey from December 2017 [27], it was one of the most used Web servers in 2017 and we have had experience with this server.

The Dockerfile was created in a similar way to the backend, by using Node image from Docker Hub[12] to download dependencies and run the “docker build –prod” command;

[9]https://hub.docker.com/_/openjdk/
[10]https://hub.docker.com/_/maven/
[11]https://nginx.org
[12]https://hub.docker.com/_/node/

and then using the NGINX image[13], by copying files generated by the build command into this image, to be served.

The usage of this Dockerfile in the Docker Compose file is similar to how the Dockerfile on backend was used.

[13]https://hub.docker.com/_/nginx/

7 Testing

According to Sommerville [1] the testing process has two goals: to demonstrate to the developer and the customer that the software meets its requirements and discover situations in which the behaviour of the software is incorrect. In this section we will go through the testing process we have followed in order to achieve these goals. Firstly the static code inspections and unit testing are presented; followed by integration and system tests; finally acceptance testing during meetings with the representatives of the stakeholders to validate and verify a current version of the system.

7.1 Static Testing

Static testing has been done at earlier stages of the development process. During this kind of testing the code was not executed but analysed. As long as the software has been developed, the development group has reviewed it in plenum. Some parts of the source code has been reviewed during the Sprint Review Meetings with the representatives of the stakeholders. Ideas and suggestions arose during all these reviews, that allowed us to improve the quality of the source code. As we mentioned in section 4.3.4 we have used IntelliJ as a development environment. IntelliJ has code analysis tools that inform about errors and certain code inefficiencies. These analysis tools has been used while the code has been developed; also to run bulk inspections in order to focus the analysis on certain parts of the code.

7.2 Unit Testing

Unit testing aims to discover bugs in the software during development. It consists of testing program components, such as e.g. methods [1].

All test classes (see listing 7.2) are annotated with @RunWith(SpringRunner.class)[1] which cuases that JUnit uses the Spring test framework. They are also annotated with @SpringBootTest[2] which makes sure that all the Spring Boot features are loaded.

For the unit tests, a separate "application.properties" file is created, which is loaded with the @TestPropertySoruce annotation; this is done to tell Spring to use a different database (to avoid operations on the real database instance). We use a H2 in-memory database[3] for unit testing.

Since an in-memory database is used, we need to fill it with initial data every time we create the database, to achieve this a Utils class is created with a function annotated @PostConstruct, this annotation makes the function run right after constructing the class object in order to initialise static variables. This function listing 7.1 adds pre-defined message types, actor types and domain to the database.

[1]https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context /junit4/SpringRunner.html

[2]https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/context/SpringBootTest.html

[3]http://www.h2database.com/html/main.html

```java
@PostConstruct
private void initStatic() {
    actorRepository = this.actorRepository0;
    messageTypeRepository = this.messageTypeRepository0;
    actorTypeRepository = this.actorTypeRepository0;
    domainRepository = this.domainRepository0;
    adminRepository = this.adminRepository0;

    // Initialize database with message types
    messageTypes.add(smsMessageType);
    messageTypes.add(internalMessageType);
    messageTypes.add(emailMessageType);
    messageTypeRepository.save(messageTypes);

    // Initialize db with actor types
    actorTypes.add(appActorType);
    actorTypes.add(userActorType);
    actorTypeRepository.save(actorTypes);

    // Initialize domain
    domainRepository.save(domain);
}
```

Listing 7.1: Extract of Utils PostConstruct

An example of unit testing for business logic (adding an Actor) is presented in listing 7.2. It uses the Utils class, created to mock different objects necessary for testing (in this example: an ActorPayload and a currently inlogged user - App Admin). The test checks if the object created is not null, and that the name of the actor is the same as indicated in the payload.

```java
@RunWith(SpringRunner.class)
@SpringBootTest
@TestPropertySource(locations = "classpath:application-integrationtest.properties")
public class ActorServiceTests {
...
    @Test
    @Transactional
    public void addActorTest() {
        ActorPayload actorPayload = Utils.createMockActorPayload();
        Actor actor = actorService.add(Utils.createMockUser().getUserId(), actorPayload);
        assertNotNull(actor);
        assertThat(actor.getName()).isEqualTo(actorPayload.getName());
    }
...
}
```

Listing 7.2: Extract of ActorServiceTest

The business logic of MaaS is implemented by services, therefor the main focus, considering test coverage, has been on classes from the service package. Following the requirements (appendix J.12.1) the test coverage on main functionality should be over 75% or explained if below that level; as presented below (fig. 25) all services except for KeyCloakUserService are over the expected level; that service is responsible for fetching data from external API (Keycloak) and extracting data from HTTP Request to MaaS API (authorisation token as described in section 5.3.1), therefor we used integration testing (section 7.3.2) instead of unit test for this service.

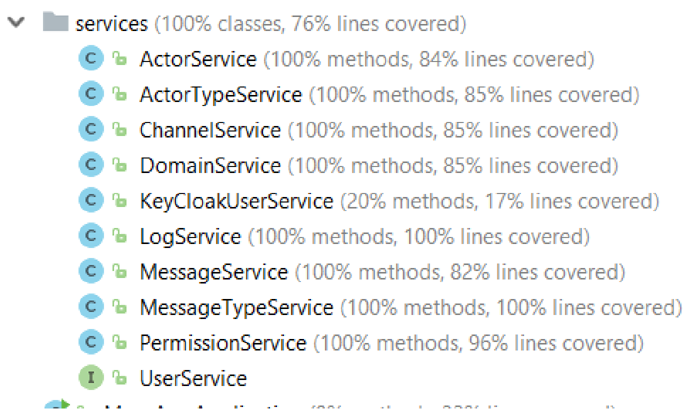

services (100% classes, 76% lines covered)
ActorService (100% methods, 84% lines covered)
ActorTypeService (100% methods, 85% lines covered)
ChannelService (100% methods, 85% lines covered)
DomainService (100% methods, 85% lines covered)
KeyCloakUserService (20% methods, 17% lines covered)
LogService (100% methods, 100% lines covered)
MessageService (100% methods, 82% lines covered)
MessageTypeService (100% methods, 100% lines covered)
PermissionService (100% methods, 96% lines covered)
UserService

Figure 25: Test coverage, IntelliJ

7.3 Integration Testing

In addition to testing of a single unit we run tests of integration of multiply units working together.

There are tests for querying the API directly through HTTP requests (see listing 7.3). This creates a mock sender and receiver, and uses TestRestTemplate[4] to send the HTTP request.

7.3.1 White Box Testing

Knowing how the data for the API response is created (described previously in section 5.5.5) it has been possible to prepare an expected payload and compare it with the one generated by a POST Request. In the following example of sending a message (listing 7.3) two actors (App), having access to an Internal channel, are created in the test environment. Based on the Internal message flow in the system (as presented previously in section 5.5) a confirmation of a successfully sent message should have the JSON format that can be compared with the expected payload.

```java
@Test
public void sendInternalMessageOneOkTest() {
    Actor senderActor = Utils.createMockActor();
    Actor receiverActor = Utils.createMockActor();

    SendMessageResponsePayload expextedPayload = new SendMessageResponsePayload(
            Arrays.asList(new SendMessageResponsePayload.ReceiverDetails(
                                    200,
                                    receiverActor.getName(),
                                    "OK"
            )),
            "Found all receivers",
            200
    );
    OutboxMessagePayload outboxMessagePayload = Utils
        .createMockInternalOutboxMessagePayload(senderActor, receiverActor);

    SendMessageResponsePayload payload = this.restTemplate.postForObject(
            "/msg/api/send", outboxMessagePayload, SendMessageResponsePayload.class
    );

    assertThat(payload).isEqualTo(expextedPayload);
}
```

Listing 7.3: Testing send message API

[4]https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/web/client/TestRestTemplate.html

7.3.2 Black Box Testing

As described previously by using an ActorRepository example (section 5.4.4), Spring Data Repositories are utilised in the system. It has been unknown for us how the specific SQL queries were created and if their results are as expected; to ensure the correctness of @Entity class definition (including SQL constraints) and query methods, phpMyAdmin was used to inspect the results of different API calls during other tests. The main goal for those tests in the project has been to establish a database structure that corresponds with the design model and confirm that services (that call query methods) are able to store expected data in the database.

Another tool that was used to perform integration tests of backend API was Postman. In the development environment (with both Keycloak and database instances running, or not in test cases of unavailable services) we simulated different API calls and inspected responses; also in those cases it was possible to extend those tests with database inspections using phpMyAdmin. To test the integration of Keycloak and a proper behaviour of the KeyCloakUserService we extracted authorisation token from the HTTP Request from Admin Panel in-logging screen and reused it (in both original and altered forms). Additionally we could operate directly on the Keycloak instance using its Admin Console; altering user's (App Admin) rights and force logging out are some of the examples.

7.4 System Testing

After the previous tests were conducted it was possible to implement new functionality into frontend and test the current version of the entire MaaS system through the Admin Panel web interface.

7.5 Validation and Verification

During the Sprint Review Meetings, the Product Owner could check the functionality on different stages of the development process. The aim of these tests were not to find code bugs, but to check if the right product and functionality were developed; additionally the code quality could be verified. [1].

The Review Meeting after the fifth Sprint (appendix J.10.1) is an example of the Validation and Verification process that was present during development. When the current stage of the MaaS system (following requirements in the Sprint Backlog) was demonstrated we confirmed that it is the right product, but new ideas, suggestions and requirements arose. Below some of the adjustments that have been done to follow the new and more precise requirements:

- **Timestamp for log of messages:** We presented a graphical interface that showed if a message had been delivered or not. The Product Owner discovered that it would be useful for inspections purposes to have a timestamp, indicating when a message was received and delivered. During planning of the sixth sprint (appendix J.10.2) those suggestions were considered in the Sprint Backlog.
- **Admin inspection functionlity:** According to requirements, both the App Users and the App Admins have access to App message inbox. Whenever a message is delivered to an actor it should be marked as delivered. However when we showed and tested this functionality with the stakeholders, we found out the following: If

an App Admin inspects the message inbox of an App, then those messages were marked as delivered since the admin panel used the same API endpoint as an App. This could led to the uncertainty of who has fetched the App inbox, an App User or an App Admin. A new requirement from Product Owner was added after that testing: an "inspection mode" that allows App Admin to inspect the inbox of an App, but at the same time the messages are not marked as delivered. To fulfil this requirement a new API endpoint was implemented; this new API, would detect if the request came from the App or from the App Admin. In the last case the messages are not marked as delivered.

A picture from Development Team meeting (fig. 26) shows how that issue was handled.

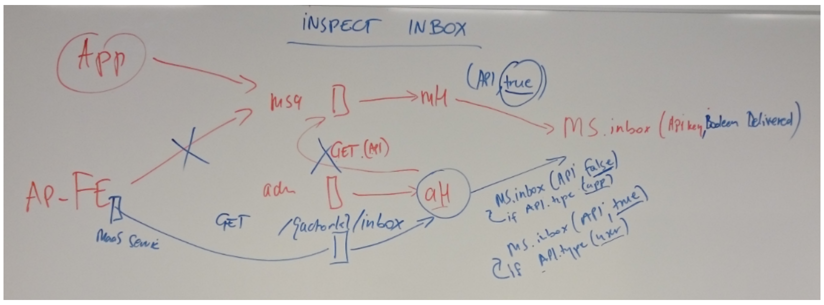


Figure 26: API changes after Alpha Testing

As mentioned previously (section 7.1) we inspected the source code during Review Meetings. When the system functionality was demonstrated and validated after the fifth Sprint, we wanted to present concrete implementation choices and validate the code regarding its quality and chosen technology. Our concern at that point was the difficulty to write good unit tests that could check business logic without unnecessary dependencies. Following the advice to keep Controllers as small as possible, we decided to refactor the structure of the code as follows:

- Firstly, we extracted all remaining functionality from all methods annotated with @RequestMapping; those were supposed to receive a HTTP Request, delegate work, and send HTTP Respond base on results of that delegated work.
- Secondly, the entire business logic of the system was gathered and a Service layer (with defined boundaries towards both Controllers and Database Management layers) was established.
- Finally the business logic was divided into several entities with a common concern, those became the concrete service classes.

On the following Review Meeting we presented both new structure and new unit tests, that corresponded with it.

7.6 User Testing

The purpose of this project was to develop a REST API service that would be integrated with various client systems (Headit customers); those system are supposed to be integrated later by a third part. Deployment infrastructure was not available during development, so the tests were limited to the development environment. The project hasn't reached the stage for wider use of User Testing.

"Alpha Test" and "Beta Test" were not relevant in the context of the project maturity.

8 Discussions and Conclusions

In previous chapters we have presented the solutions that have been chosen, in this one we want to present reasons for the final decisions and discuss alternative solutions that were considered during the development process. Later we evaluate our work during the project and how the work progress differs from the one we planned in the beginning. The chapter is concluded with the presentation of a potential future development and integration of the system; the final section of this chapter presents the summarised results of the project, concerning the product, how it answers requirements and what we have learnt.

8.1 Choices

In this section we discuss choices made during the whole development process; this includes both technology and organisational choices.

8.1.1 HTPP vs HTTPS

Already during the first meeting with the representatives of the stakeholders we discussed the confidentiality of sent messages (appendix J.2). One of the issues was use of HTTPS. We agreed with Product Owner's proposal to start development using HTTP. It has been decided that before the system is to be put into production the SSL certificate need to be acquired, HTTP ports closed and only HTTPS access to API allowed.

8.1.2 Securing API with Keycloak

As it was mentioned in section 5.3.1 the frontend is secured with Keycloak. The same policy is applied to backend as it was described in section 5.4.1. If an App Admin wants to create, administrate or delete an App, he or she must be authenticated by Keycloak. Several solutions were considered to secure this part of the API offered by MaaS, among them Spring Security. On the Sprint Review Meeting, after the third sprint (appendix J.8), we presented Keycloak integration with Spring and it was accepted without need to change security model to the one including Spring Security framework. The Spring Security framework allows to secure an existing application, so it could be applied in the future if required; during the discussions with the Product Owner it was decided that we should focus on the main functionality of the system.

API endpoints used by an App to access MaaS resources (send a message, fetch own inbox) are not secured with Keycloak. In the beginning of implementation (the first sprint dedicated to programming) we met problems with obtaining from Keycloak the "offline token", that was supposed to identify an App Admin, and an application that could utilise it and connect to MaaS. Instead of creating a mock application that could be used as a sender (App) we started the development of the core MaaS functionality based on an open API endpoint for message sending and available tools for sending HTTP Requests (Postman). After that sprint we presented and discussed a proposal to use UUID as API access keys, this was accepted (appendix J.8) and used since then, App verification process is described previously in section 5.5.3. The reason to not securing a message API with Keycloak is that an application would have to store additional App Admin credentials (e.g. in form of an offline token, that represents a Keycloak registered user, without need to store username and password). Those credentials could be potentially misused to get access to the API protected by Keycloak (by an App pretending to be an App Admin). Additionally, in the case of an application that is administrated by more than one App Admin, it would have to store multiply (or selected) credentials and send a message as one of administrators not as an application itself, so additional identification of sender would also be necessary.

8.1.3 Development Methodology

Already at the first meeting with the representatives of the stakeholders (during the project presentation at NTNU) it was clear that the scope, the priority focus, requirements etc. might be changed during the time dedicated to the project. Due to the lack of experience in some of the technologies, that were planned to be used (Spring, Keycloak, Angular among them) and integration between all components (both developed and reused), it would be difficult to present a reliable project plan that would guarantee the delivery of the expected product. We decided to hold regular (at least once in two weeks) meetings with the representatives of the stakeholders to discuss a current progress, test solutions we found since the last meeting and plan the next period of work; the presented (partial) solutions were to be developed incrementally (including code improvement) considering potential requirements change or problems blocking the progress. The representatives of the stakeholders were supposed to be involved in the progress although the final decisions on chosen technology or solution were left to us.

Based on the following description of the work progress we decided to use an Agile methodology for the development process. The Scrum methodology was known for both parts and we decided that it would fit the project. Following the methodology a Product Owner was chosen and we focused on creating an initial Product Backlog.

Evaluation of the chosen methodology later in section 8.2

8.1.4 Backend Framework

Regarding the choice of framework or technology to develop the backend, the only requirement from the Product Owner was to use well known and open source technologies (organisational requirements described previously in section 2.4.2). The Product Owner suggested Spring Boot but we had the possibility to choose another framework. We decided that Spring Boot could be a good choice based on:

- **Java:** we wanted to work with a framework that uses a programming language we had learnt and worked with before.
- **New technology:** during this project we had to learn new technologies, and we found convenient to choose a Java framework as one of them.
- **Easy start:** we found that Spring Boot is well documented, has a lot of tutorials and support for starting a new project.
- **Keycloak integration:** another positive aspect of Spring Boot was the fact that it could be integrated with Keycloak (Keycloak offers an adapter for Spring Boot [14]).

8.1.5 Database

Concerning the election of database technology, we evaluated the advantages and constraints of SQL and NoSQL databases. We discussed this with Product Owner (appendix J.7) and he suggested to study the example of a "blog entry" in case we wanted to use a NoSQL database. We appreciated the flexibility that NoSQL offers, but we decided to use a relational database due to the following reasons:

- **Experience:** courses we took during the studies concerned mostly SQL databases.
- **Time:** to get more familiar with the NoSQL technology and integration of it into the project would take time and effort that we decided to put into the other aspects.
- **Integration:** we chose a Spring framework and found more resources, that explained how it could be integrated with SQL database by using JPA.

8.1.6 Frontend Framework

We had experience with plain HTML, CSS, JavaScript from courses, but wanted to learn a framework that would bind all those together. The only framework we had experience with is Polymer[1], but we wanted to learn a new one. As for the backend, a consideration of a known and open source technology was requested. The Product Owner suggested Angular 5 (organisational requirements described previously in section 2.4.2) and we decided to try it. We found that Angular framework provided a clear separation between the logic (TypeScript) and the view (HTML) which helped us to hold control; it was straight forward and intuitive to build an Angular project thanks to the component based structure. As for the negative sides of Angular, it was not so easy to implement the testing functionality.

8.1.7 Channels

The first model of MaaS, based on the initial project description, presumed that messages will be forwarded to all channels in a similar way, and additional messages might be sent between two users through the Admin Panel. During the meetings with Product Owner we discussed priorities and it has been decided, that the extended interpretation of internal messages (can be exchanged between two or more Apps) should be implemented first; the complexity of the system increased due to the necessity of handling new functionality (e.g. a message inbox stored and accessed beyond presumed log only, delivery status etc.).

The effort put into internal message exchange caused that the time dedicated to other channels was reduced, but we decided that we wanted to implement at least two outgoing channels that would support:

- a possibility of sending a message to an mobile/desktop application that the Recipient may use (PUSH to mobile realised using a webhook as described in section 5.6)
- a common channel that doesn't require any additional software (implemented email sending).

After we have made research regarding SMS channel, it turns out that both binding to the chosen provider and a significant cost is required in order to use the provider API. We discussed our findings with Product Owner and decided not to implement this

[1]https://www.polymer-project.org

channel as bound to a specific provider, but instead we added a handler for an exception (ChannelNotImplementedException) thrown by a "send" method in SmsChannel class; this is to be changed when a provider will be chosen.

8.2 Work Evaluation

In this section we will describe the impressions we have about the work and the consequences of choices made during the project realisation.

8.2.1 Scrum

First of all, we think that is has been positive to use the Scrum methodology for the following reasons:

- **Sprints:** on every Sprint we presented the solutions to all the tasks that were previously agreed with the Product Owner during the Sprint Planning Meeting. The fact of having milestones every two weeks helped us to have control over the project and motivated us to work towards the goal.
- **Feedback:** we found out that it was motivating to get a frequent feedback from the Product Owner; it helped us to remain on track and gave inspiration and ideas too.
- **New Requirements:** some new requirements arose during the project. We think that the methodology helped us to deal with them.

8.2.2 New Technologies

One of the goals we had when we started the project was to learn and get as much experience as possible. We think we have achieved this goal and all group members have worked with all aspects of the project. We have gained experience with Spring Boot to develop the REST API. We have used JPA and Hibernate to work with the database. Angular 5 was used to develop a Single-Page Application (Admin Panel). Additionally we have used Maven and Docker, for build and deployment.

8.2.3 Risk Management

During the project we have experienced some problems that were identified already when we created the project plan (appendix F.5.3); those risks were described and addressed.

- **Time needed to learn new technologies:** To overcome this problem we assigned the "specialist" (in the specific technology) role to one group member. A "specialist" had responsibility to research and learn a tool/technology and share his knowledge with the others. The idea was to speed up the learning process during the project.
- **Some task needed more time than expected:** We had god communication about eventual delays and good will to make efforts to get the different tasks done.

8.2.4 Comparison of Plan and Realisation

The initial plan for implementation (appendix F.6) and the actual work progress (appendix B) differs. The following tasks were realised differently according to changing requirements and difficulties that occurred during development:

- Sender/Receiver mock application were not implemented separately; the functionality of sending/receiving messages in Admin Panel and existing tools (Postman) were used.

- Web-app development took one extra sprint due to extended functionality (replacement of Sender/Receiver mock application) and additional work with REST API.
- REST API service was developed separately from Web-app Angular project; therefor in the Gantt diagram that represents the actual progress (appendix B) time dedicated to REST API (Message, Admin) is presented separately, not as a part of Admin Web-app that we planned in the beginning.
- Admin mobile application was not realised; already during the first meeting (appendix J.2) we discussed necessity of a mobile version of Admin Panel and followed required focus on a web-application for PC.
- Cloud Deployment task has been partially realised (described previously in chapter 6); we have used Docker technology from the beginning of the development and prepared the project for potential future deployment, but since the Cloud provider hasn't been chosen and the deployment infrastructure not available we could not test (and potentially adjust) MaaS outside of the development environment.
- Report - we started to work with the final report earlier than expected and therefor an implementation report were not written as a separate task. Additionally the third status report was replaced with the submission of the main part of this report.
- Outgoing Channels - we have dedicated an additional sprint for work with the outgoing messages; internal messages we focused on in the fifth sprint required more effort than expected, so additional work on Slack and Email channels was delayed.

8.3 Critique of Bachelor Thesis

In this section we would like to present alternatives to the methods and solutions chosen during the work on the project, considering those we could realise differently (and better) then we initially assumed.

8.3.1 Database Technology

As described previously (section 8.1.5) we chose a relational database model. It was convenient to use in the early stage of the project, but when we reached the point when details about implementation of concrete outgoing channels were discovered we could benefit more from the flexibility of the NoSQL technology. At this point it was more efficient to remodel the existing database structure than introduce changes in both Database Management and Database layers on backend; we think that following the initial choice was a right thing to do, although it could have been better to work with NoSQL from the beginning.

8.3.2 Test on Frontend

As long as we developed the frontend, we tested it using the browser console and in addition we did functional testing as it is described in section 7.5. However we decided to run unit tests in the frontend when we already had developed a part of its functionality. We studied Jasmine[2] (a testing framework that supports Behaviour Driven Development) and Karma[3] (a tool that allows to run the test in different browsers) [28]. When we

[2]https://jasmine.github.io/

[3]https://karma-runner.github.io/2.0/index.html

started to implement them into the Angular project, the results were not easy to interpret and we weren't able to solved those issues within the time we could dedicate to. Since we didn't get the expected results with Jasmine and Karma, we decided not to put more efforts to it. We continued testing the frontend using the console and doing functional testing.

8.3.3 Time Estimation

We believe that the lack of experience caused underestimating of both complexity and time needed to perform various tasks during development. Estimations during daily Scrum meetings were not precise enough and often additional overtime needed to fulfil the tasks we decided to be done for the Sprint; regardless of the time it would take. Although some of the tasks were completed earlier than evaluated, the total workload for each Sprint was not balanced; we should rather put more issues back to the Product Backlog after each Review Meeting and prepare the development plan more carefully.

8.3.4 Tasks Order

Additionally to the time estimation on the daily work basis, we could have also done a better long time work evaluation. From the final product perspective it might have been better to focus on REST API service on backend and when it would become mature enough start with the frontend project to create a web interface for administration using already implemented APIs. The choice we made at the beginning caused that we had to handle all unknown technologies (Angular, Spring, Keycloak) at once; one of the effects was the time loss due to refactoring and integration issues. On the other hand it might have been difficult to present and validate a current progress to the Product Owner based on code reviews and development tools like Postman.

8.4 Future Work

We believe that the system is mature enough to demonstrate it to end-users and the functionality it provides could result in the integration of MaaS in the systems they already use or planning to order. However to go over to the production it is necessary to address all security issues (e.g. those discussed in section 8.1).

Another technical challenge might be a deployment; although we prepared the system so it should be possible to deploy it using Docker technology supported by current providers of Cloud services, some additional adjustments might be necessary when the provider will be chosen.

To allow better integration with the current system (e.g. the one presented in the business case in appendix C.1) it is recommended to decide on SMS provider and implement the provider specific payload into SmsChannel class. Adding a possibility to receive a message through another common channel (besides implemented email channel) like SMS would significantly increase attractiveness of the system for potential customers.

It is possible to build a wider system based on MaaS; e.g. a client application (desktop or/and mobile) that could be used as App for sending and receiving messages; and potentially configure own settings through API requests to MaaS (similar to Admin Panel, but considering limited rights).

The system has been designed and implemented so it should be easy to add new outgoing channels. Since the only limitation is an Internet connection it is actually possible to exchange messages between any devices one could imagine; so the IoT is available to integrate with MaaS.

8.5 Results

In this section we present our interpretations of the results and the development process.

8.5.1 Message Service

We manage to develop a service that provides API for message sending and access configuration. Messages are forwarded and logged on all stages of processing. The messages processed by MaaS can be received by three types of channels:

- Internal messages - support for communication between registered applications within a domain.
- Receiver dependent - a webhook provided by Slack, that allows to receive a message in an application already used by Recipient.
- Common - a channel that does not require registration in MaaS or a specific application to receive a message. Email is an example of this channel.

We developed a simple web interface, to contact MaaS and administrate Apps, but it is possible also to call API directly; this could allow integration into an external administration interface.

8.5.2 Efficiency and Performance Targets

Since the system hasn't been in production, it is difficult to evaluate, if it would reach expected goals. It works as a stand-alone application and REST API that the system provides is possible to integrate with external services; so performance targets, as named in the project plan (appendix F.1.2), are fulfilled. Considering efficiency goals the system hasn't been yet integrated or presented to Headit customers, so it is too soon to evaluate benefits they gain.

8.5.3 Learning Goals

The main goal, to experience a full-stack developer work, as named in the project plan (appendix F.1.2), was achieved, as we have worked with all aspects of a complete system. We had an opportunity to try and learn new technologies. Not all of them were easy to implement, but fails and integration issues allowed us also to learn what to avoid.

We became more familiar with Agile methodology (Scrum) in both overview aspect during process management, and daily work with Scrum meetings etc.

Our programming skills have been improved as we implemented the designed system model; we could measure a constant progress during development by evaluating discussions considering design and implementation improvement along the entire process, as we discovered new possibilities. The main technologies that we could name are Spring, Angular 5 frameworks and database integration using JPA.

Bibliography

[1] Sommerville I. Software Engineering. Pearson Education Limited; 2016.

[2] Connolly T, Begg C. Database Systems. Pearson Education Limited; 2015.

[3] Jendrock E, Ball J, Carson D, Evans I, Fordin S, Haase K. The Java EE 5 Tutorial. Oracle; 2010.

[4] Richards M. Software Architecture Patterns: Understanding Common Architecture Patterns and when to Use Them. O'Reilly Media, Incorporated; 2015.

[5] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Addison-Wesley; 1994.

[6] Baeldung. Build your API with Spring [Internet]. Baeldung; [cited May 2018]. Available from: http://www.baeldung.com.

[7] Fowler M. Inversion of control containers and the dependency injection pattern [Internet]. Martin Fowler; 2004 [cited May 2018]. Available from: https://martinfowler.com/articles/injection.html.

[8] Oracle. Core J2EE Patterns - Data Access Object [Internet]. Oracle; 2018. Available from: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html.

[9] Google Inc. Angular Docs [Internet]; [cited May 2018]. Available from: https://angular.io/docs.

[10] Agile Alliance. Scrum Master [Internet]. Agile Alliance; [cited May 2018]. Available from: https://www.agilealliance.org/glossary/scrum-master.

[11] Atlassian. What is Git [Internet]. Atlassian; [cited May 2018]. Available from: https://www.atlassian.com/git/tutorials/what-is-git.

[12] Keycloak. About Keycloak [Internet]. Keycloak; [cited May 2018]. Available from: https://www.keycloak.org/about.html.

[13] Keycloak. Keycloak Documentation [Internet]. Keycloak; [cited May 2018]. Available from: https://www.keycloak.org/docs/3.0/index.html.

[14] Keycloak. Securing Apps [Internet]. Keycloak; 2018. Available from: https://www.keycloak.org/docs/latest/securing_apps/index.html.

[15] Vigolo MG. Keycloak for Angular [Internet]. NPM Inc; [cited May 2018]. Available from: https://www.npmjs.com/package/keycloak-angular.

[16] Bernard E. Hibernate Annotations [Internet]. JBoss; [cited May 2018]. Available from: https://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/.

[17] Docker Inc. What is a container [Internet]. Docker Inc; [cited May 2018]. Available from: https://www.docker.com/what-container.

[18] aplatypus. Ticket #16801 Running a Virtual Machine when Windows Hyper-V is enabled should NOT Crash Windows [Internet]. Virtualbox; [cited May 2018]. Available from: https://www.virtualbox.org/ticket/16801.

[19] Docker Hub. mariadb - Docker Hub [Internet]. Docker Inc; [cited May 2018]. Available from: https://hub.docker.com/_/mariadb/.

[20] phpMyAdmin. phpmyadmin - Docker Hub [Internet]. Docker Inc; [cited May 2018]. Available from: https://hub.docker.com/r/phpmyadmin/phpmyadmin/.

[21] Docker Inc. Networking in Compose [Internet]. Docker Inc; [cited May 2018]. Available from: https://docs.docker.com/compose/networking/.

[22] JBoss. keycloak - Docker Hub [Internet]. Docker Inc; [cited May 2018]. Available from: https://hub.docker.com/r/jboss/keycloak/.

[23] Docker Inc. Use multi-stage builds [Internet]. Docker Inc; [cited May 2018]. Available from: https://docs.docker.com/develop/develop-images/multistage-build/.

[24] Docker Inc. Best practices for writing Dockerfiles [Internet]. Docker Inc; [cited May 2018]. Available from: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

[25] Docker Inc. Compose file version 3 reference [Internet]. Docker Inc; [cited May 2018]. Available from: https://docs.docker.com/compose/compose-file/.

[26] Google Inc. Angular 5 - Deployment [Internet]. Google Inc; [cited May 2018]. Available from: https://angular.io/guide/deployment.

[27] Netcraft. Angular 5 - Deployment [Internet]. Netcraft; [cited May 2018]. Available from: https://news.netcraft.com/archives/2017/12/26/december-2017-web-server-survey.html.

[28] Murray N, Coury F, Lerner A, Taborda C. Ng-Book. The complete guide to Angular. Fullstack.io; 2018.

[29] Oracle. Code Conventions for the Java TM Programming Language [Internet]. Oracle; [cited May 2018]. http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html.

A Definitions

Text	**Description**
MaaS	Message as a Service
CRUD	Create, Read, Update, Delete
UUID	Universally Unique Identifier
POJO	Plain Old Java Object
MaaS Admin	A superuser that administrates the MaaS server.
App Admin	An administrator for a group of applications using the MaaS system.
Actor	Represents a Receiver or a Sender using our API (App).
App	An application registered at MaaS, eligible to use the service.
App User	An user of an application connected to the MaaS service.
Receiver	An application, hardware (e.g. mobile phone) or other interface the out-going message is send to, from the MaaS server
Recipient	An user of a Receiver
PMA	phpMyAdmin
IoT	Internet of Things
API	Application Programming Interface

B Gantt Diagram

The Gantt diagram presented below on figure 27 shows how the development process was realised.

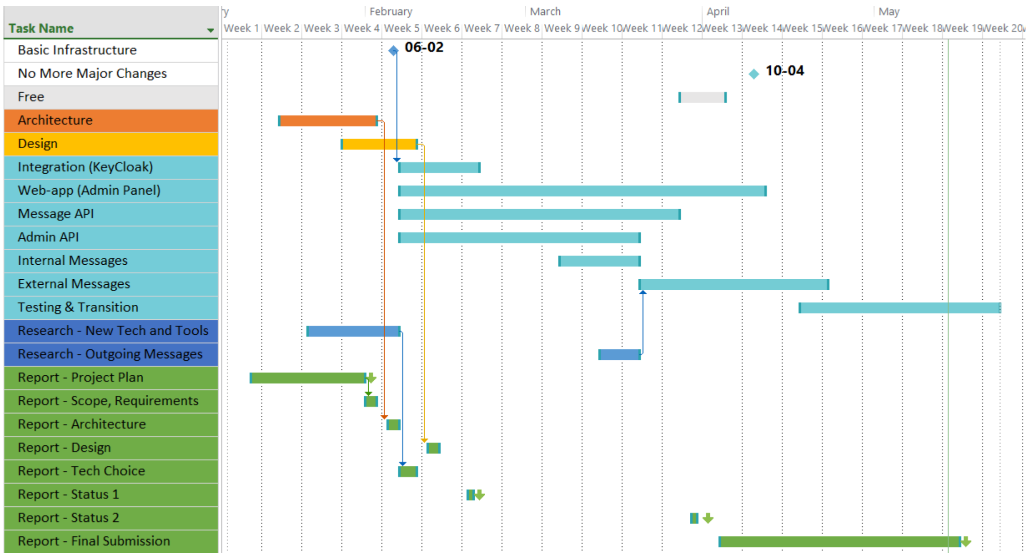


Figure 27: Gannt diagram for project progress, MS Project

B.1 Milestones and Deadlines

- Project Plan - 01.02.2018
- First status report - 20.02.2018
- Second status report - 01.04.2018
- Final submission - 16.05.2018

Additionally we have chosen two deadlines to insure the completion of both working software and final report.

- Infrastructure - 06.02.2018 - to ensure that by the start of the third sprint we would have all necessary tools and resources to start with programming activities.
- No more major requirement changes - 10.04.2018 - We agreed to not accept major changes beyond that point (start of seventh sprint) in order to reliably deliver the software defined so far.

C User Stories and Scenarios

C.1 Business case - My Custom Suit

My Custom Suit delivers custom-made suits. It controls a chain of shops and each of them offers tailoring services self and (or) has an agreement with tailors working in their own studios.

Currently ordering customer service has access to a web-application where the tailors and accounting office can register and check the order status.

The order flow in the system works as follows:

- A customer visits a shop and chooses a suit model.
- Measurements are taken and a delivery date is set.
- The order is sent to the tailors.
- The tailor informs when the work is done.
- When all components are ready, the customer is informed that the suit is ready to be tried.
- All the components are adjusted and sewed together.
- The customer picks up the suit.
- The tailors receive the payment for the components they sewed.

My Custom Suit noticed that more orders come in some periods of the year. It is important for them to address them all and inform the customer quickly, regardless of number of orders. This would build the company's reliability. It is important that punctuality is supported on all the stages of the order realisation.

C.1.1 Actors involved

- Customer Service. Works in the shop, registers measurements, registers an order in the system, edits the order in the system after a suit is tried and completes the order.
- Customer. Orders a suit.
- Tailor. Sews part(s) for a suit.
- Accounting Office. Sends payment to tailors.

C.1.2 Goals

Automatic information to customer

The current process is manual. The customer service checks an order and if all the components are ready contacts the customer via email or sms. Automation of the process could let the customer service work with other tasks.

- More effective communication with customers.
- Avoid waiting too long to try a suit when all components are ready.

Effectiveness in completing the order

Currently all the information about the suit components is gathered manually. It can happen that the components must wait even if they are ready due to the information didn't come or wasn't fetched. This process can be enhanced by automatic warning when a component is ready. This automation can also increase the production. When all components are ready the customer should be informed to come to try the suit.

- Reduce storing time of ready components.
- More effective production of complete suits.

Assumption: A tailor has access to a computer or mobile phone.

D Code Examples

This appendix consists of code examples, that were too large to present as a whole in the report. Parts of functions/classes in this appendix are placed previously as listings.

```java
public SendMessageResponsePayload send(Actor actor, OutboxMessagePayload payload) {

    String content = payload.getContent();
    if (content == null || content.isEmpty()) {
        throw new WrongPayloadException("Empty message content is not allowed");
    }

    MessageType type = messageTypeService.get(payload.getType());
    if (type == null) {
        throw new WrongPayloadException("Message type not found");
    }

    SendMessageResponsePayload customResponse = new SendMessageResponsePayload();

    Message message = new Message();

    if (type.getId().equals("Internal")) {
        Set<Actor> receivers = getReceivers(actor.getDomain().getId(),
                                payload.getReceivers(), customResponse);

        if (receivers.isEmpty()) {
            throw new WrongPayloadException(
                "Empty receivers list is not allowed for type:" + type.getId());
        }

        message.addReceivers(receivers);
    } else {
        customResponse.setCode(200);
        customResponse.setMessage("External message accepted for processing");
    }

    message.setType(type);
    message.setContent(content);
    message.setSender(actor);
    message.setSubject(payload.getSubject());

    message.setRecipients(getExternalRecipients(payload.getRecipients(), message));

    MessageWithMetadata messageWithMetadata =
        getMetadata(payload.getConfigId(), payload.getMeta(), actor.getId());

    messageWithMetadata.setMessage(message);
    setDefaultDataFromMetadata(messageWithMetadata);

    message.addLog(new Log(message, "Message received by MaaS API"));

    sendMessageToProcessing(messageWithMetadata);

    return customResponse;
}
```

Listing D.1: Send message function from MessageService class

```java
private Set<Actor> getReceivers(int senderDomainId, List<String> requestedReceivers,
SendMessageResponsePayload customResponse) {
    Set<Actor> receivers = new HashSet<>();

    if (requestedReceivers == null) {
        return receivers; // Empty set.
    }

    for (String receiver : requestedReceivers) {
        Actor actorReceiver = actorRepository.findActorIfActiveByName(receiver);
        if (actorReceiver != null) {
            if (actorReceiver.getDomain().getId() == senderDomainId) {
                if (actorReceiver.canUseInternalChannel()) {
                    receivers.add(actorReceiver);
                       customResponse.addReceiver(200, receiver, "OK");
                } else {
                    customResponse.addReceiver(400, receiver,
                    "Actor cannot use Internal channel");
                }
            } else {
                customResponse.addReceiver(400, receiver, "Actor not in sender domain");
            }
        } else {
            customResponse.addReceiver(400, receiver, "Actor not found");
        }
    }

    if (receivers.isEmpty()) {
        customResponse.setCode(400);
        customResponse.setMessage("Haven't found any of " + requestedReceivers.size()
        + " receivers");
    } else if (receivers.size() != requestedReceivers.size()) {
        customResponse.setCode(200);
        customResponse.setMessage("Found" + receivers.size() + " of "
        + requestedReceivers.size() + " receivers");
    } else {
        customResponse.setCode(200);
        customResponse.setMessage("Found all receivers");
    }

    return receivers;
}
```

Listing D.2: Function “getReceivers” from MessageService class

```java
private MessageWithMetadata getMetadata(Long configId, Object extras, String actorId) {
    MessageWithMetadata messageWithMetadata = new MessageWithMetadata();
    messageWithMetadata.setPayloadExtras(extras);

    if (configId == null || configId == -1) {
        return messageWithMetadata;
    }

    ConfigSet configSet = this.configSetRepository.
    findByIdAndChannelRights_ChannelId_Actor_Id(configId, actorId);
    if (configSet == null) {
        throw new WrongPayloadException("Unknown config ID");
    }
    messageWithMetadata.setConfigSet(configSet);
    return messageWithMetadata;
}
```

Listing D.3: Function “getMetadata” from MessageService class

```java
private void setDefaultDataFromMetadata(MessageWithMetadata messageWithMetadata) {
    Message message = messageWithMetadata.getMessage();
    ConfigSet configSet = messageWithMetadata.getConfigSet();

    if (configSet == null) {
        return;
    }

    // Use subject in config if not set.
    if (message.getSubject() == null || message.getSubject().isEmpty()) {
        String subject = configSet.getAttributes().get("subject");
        if (subject != null) {
            message.setSubject(subject);
        }
    }
}
```

Listing D.4: Function “setDefaultDataFromMetadata” from MessageService class

```java
@Repository
public interface ActorRepository extends CrudRepository<Actor, String> {


    /**
     * Find Actor by ID
     *
     * @param id of the actor.
     * @return actor.
     */
    Actor findById(String id);

    /**
     * Check if actor exists and is active. (Without returning the actor)
     *
     * @param actorId the id of the actor to check
     * @return true or false
     */
    @Query("SELECT CASE WHEN COUNT(a) > 0 THEN true ELSE false END FROM Actor a WHERE a.id = :id
        AND (a.expire > current_timestamp OR a.expire IS NULL)")
    boolean existsActorByIdAndIsActive(@Param("id") String actorId);


    /**
     * Find if an Actor is active      *
     *
     * @param actorId actorId
     * @return Actor
     */
    @Query("SELECT a FROM Actor a WHERE a.id = :id AND (a.expire > current_timestamp
        OR a.expire IS NULL)")
    Actor findActorIfActive(@Param("id") String actorId);


    /**
     * Find if an actor is active
     *
     * @param actorName name of the actor
     * @return actor
     */
    @Query("SELECT a FROM Actor a WHERE a.name = :name AND (a.expire > current_timestamp
        OR a.expire IS NULL)")
    Actor findActorIfActiveByName(@Param("name") String actorName);


    /**
     * Deactivate an actor
     *
     * @param actorId
     * @return
     */
    @Transactional
    @Modifying
    @Query("UPDATE Actor a SET a.expire = current_timestamp WHERE a.id = :id")
    Integer deactivateActor(@Param("id") String actorId);


    /**
     * Find all Actors owned by an Admin.
     *
     * @param adminId
     * @return List of actors owned by the admin
     */
    Set<Actor> findAllByAdmins_Id(String adminId);


    /**
     * Find Active Actors owned by an certain Admin.
     *
     * @param adminId
     * @return Set of active Actors owned by the admin
     */
    @Query("SELECT a FROM Actor a LEFT JOIN a.admins k WHERE k.id = :id
        AND (a.expire > current_timestamp OR a.expire IS NULL)")
    Set<Actor> findActiveActorsByAdmin(@Param("id") String adminId);


    /**
     * Find Inactive Actors owned by an certain Admin
     *
     * @param adminId
     * @return Set of inactive actors ownned by the admin.
     */
    @Query("SELECT a FROM Actor a LEFT JOIN a.admins k WHERE k.id = :id
        AND a.expire < current_timestamp")
    Set<Actor> findInactiveActorsByAdmin(@Param("id") String adminId);

    /**
     * Finds all actor within a given domain.
     * @param domainId int domain id.
     * @return Set of actors within given domain.
     */
    @Query("SELECT a FROM Actor a WHERE a.domain.id = :id AND (a.expire > current_timestamp
        OR a.expire IS NULL)")
    Set<Actor> findActiveActorsByDomain(@Param("id") int domainId);
}
```

Listing D.5: ActorRepository interface

E REST API Documentation

This is the documentation for how our REST API works with the different endpoints, request payloads and response payloads.

The REST API documentation was created with Swagger and later exported to PDF and included in this document (See next page).

(Internal links in this document does not work, due to how including PDF works with Latex)

Overview

Api Documentation

Version information

Version : 1.0

License information

License : Apache 2.0
License URL : http://www.apache.org/licenses/LICENSE-2.0
Terms of service : urn:tos

URI scheme

Host : localhost:8080
BasePath : /

Tags

- admin-controller : Description Admin Panel Service
- index-controller : This handles all erros not handled by RestExceptionHandler
- message-controller : Description Message Service

Consumes

- `application/json`

Produces

- `application/json`

Paths

Returns information about all the admins registered in KeyCloak

```
GET /admin/api/allusers
```

Description

The user must have a rolle as Maas_Admin

Responses

HTTP Code	Description	Schema
200	OK	< CurrentUser > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

An admin adds a new actor

```
POST /admin/api/db/actors
```

Description

Notes

Parameters

Type	Name	Description	Schema
Body	**payload** *required*	payload	ActorPayload

Responses

HTTP Code	Description	Schema
200	OK	Swager example Actor

HTTP Code	Description	Schema
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Returns all the actors asociated with this user

```
GET /admin/api/db/actors
```

Parameters

Type	Name	Description	Schema	Default
Query	**status** *optional*	all, active or inactive	string	`"all"`

Responses

HTTP Code	Description	Schema
200	OK	< Swager example Actor > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Adds a new kind of actor

```
POST /admin/api/db/actors/types
```

Description

You must have admin rights

Parameters

Type	Name	Description	Schema
Body	**payload** *required*	payload	ActorTypePayload

Responses

HTTP Code	Description	Schema
200	OK	ActorType
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Returns the different kinds of actors

```
GET /admin/api/db/actors/types
```

Description

You must have admin rights

Responses

HTTP Code	Description	Schema
200	OK	< ActorType > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Returns information of a own actor

```
GET /admin/api/db/actors/{actorId}
```

Description

You must have admin rights and own the actor

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string

Responses

HTTP Code	Description	Schema
200	OK	Swager example Actor
401	Unauthorized	No Content
403	Forbidden	No Content

HTTP Code	Description	Schema
404	Not Found	No Content

Tags

- admin-controller

Edits information about a own actor

```
PUT /admin/api/db/actors/{actorId}
```

Description

You must have admin rights and own the actor

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string
Body	**payload** *required*	payload	ActorPayload

Responses

HTTP Code	Description	Schema
200	OK	Swager example Actor
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Removes an own actor

```
DELETE /admin/api/db/actors/{actorId}
```

Description

You must have admin rights and own the actor

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string

Responses

HTTP Code	Description	Schema
200	OK	ResponseStatusPayload
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

Tags

- admin-controller

Updates a channel configuration for a certain actor

```
PUT /admin/api/db/actors/{actorId}/config/{configId}
```

Description

You must have admin rights and own the actor

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string
Path	**configId** *optional*	config ID to update	integer (int64)
Body	**payload** *required*	payload	ConfigSetPayload

Responses

HTTP Code	Description	Schema
200	OK	ConfigSet
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Deletes a channel configuration for a certain actor

```
DELETE /admin/api/db/actors/{actorId}/config/{configId}
```

Description

You must have admin rights and own the actor

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string
Path	**configId** *optional*	configuration ID to be removed	integer (int64)

Responses

HTTP Code	Description	Schema
200	OK	Swager example Actor
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

Tags

- admin-controller

Configures a channel for a certain actor

```
POST /admin/api/db/actors/{actorId}/config/{typeId}
```

Description

You must have admin rights and own the actor

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string
Path	**typeId** *optional*	The kind of channel that is going to be configured. i.e. (SMS, E-mail...)	string

Type	Name	Description	Schema
Body	**payload** *required*	payload	ConfigSetPayload

Responses

HTTP Code	Description	Schema
200	OK	Swager example Actor
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Gets the inbox of a certain actor

```
GET /admin/api/db/actors/{actorId}/inbox
```

Description

You must have admin rights and own the actor

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string

Responses

HTTP Code	Description	Schema
200	OK	< InboxMessagePayload > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Returns all the logs associated with an Actor

```
GET /admin/api/db/actors/{actorId}/log
```

Description

You must have admin rights and own the actor

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string

Responses

HTTP Code	Description	Schema
200	OK	< Log > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

See the status of message

```
GET /admin/api/db/actors/{actorId}/log/{messageId}
```

Description

You must have admin rights and own the actor that sent the message

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string
Path	**messageId** *optional*	The Id of the message wanted to show log for	integer (int64)

Responses

HTTP Code	Description	Schema
200	OK	OutboxMessagePayload
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Adds a new domain to the MaaS

```
POST /admin/api/db/domains
```

Description

Only for admins with Maas_Admin role

Parameters

Type	Name	Description	Schema
Body	**payload** *required*	payload	Domain

Responses

HTTP Code	Description	Schema
200	OK	Domain
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Gets all the domains existing in the MaaS

```
GET /admin/api/db/domains
```

Description

Only for admins with Maas_Admin role

Responses

HTTP Code	Description	Schema
200	OK	< Domain > array

HTTP Code	Description	Schema
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

getAvailableInternalReceivers

```
GET /admin/api/db/domains/actors
```

Responses

HTTP Code	Description	Schema
200	OK	< InternalReceiverPayload > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Adds a admin to a certain domain

```
POST /admin/api/db/domains/{domainId}/admins
```

Description

Only for admins with Maas_Admin role

Parameters

Type	Name	Description	Schema
Path	**domainId** *optional*	ID of domain to add admins to	integer (int32)
Body	**payload** *required*	payload	Admin

Responses

HTTP Code	Description	Schema
200	OK	Admin
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Returns all the admins that belong to a certain domain

```
GET /admin/api/db/domains/{domainId}/admins
```

Description

Only for admins with Maas_Admin role

Parameters

Type	Name	Description	Schema
Path	**domainId** *optional*	ID of domain to get users from	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	< CurrentUser > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Returns the different kinds of messages (channel types)

```
GET /admin/api/db/messages/types
```

Description

You must have admin rights

Responses

HTTP Code	Description	Schema
200	OK	< MessageType > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Returns own information of the user

```
GET /admin/api/user
```

Description

Return a user by Id

Responses

HTTP Code	Description	Schema
200	OK	CurrentUser
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- admin-controller

Called on all unhandled errors

```
POST /error
```

Responses

HTTP Code	Description	Schema
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	ErrorResponse

Tags

- index-controller

Called on all unhandled errors

```
GET /error
```

Responses

HTTP Code	Description	Schema
200	OK	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	ErrorResponse

Tags

- index-controller

Called on all unhandled errors

```
PUT /error
```

Responses

HTTP Code	Description	Schema
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	ErrorResponse

Tags

- index-controller

Called on all unhandled errors

```
DELETE /error
```

Responses

HTTP Code	Description	Schema
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	ErrorResponse

Tags

- index-controller

Called on all unhandled errors

```
PATCH /error
```

Responses

HTTP Code	Description	Schema
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	ErrorResponse

Tags

- index-controller

Called on all unhandled errors

```
HEAD /error
```

Responses

HTTP Code	Description	Schema
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	ErrorResponse

Tags

- index-controller

Called on all unhandled errors

```
OPTIONS /error
```

Responses

HTTP Code	Description	Schema
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	ErrorResponse

Tags

- index-controller

Gets all channel rights associated with an certain actor.

```
GET /msg/api/config/{actorId}
```

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string

Responses

HTTP Code	Description	Schema
200	OK	< ChannelRights > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- message-controller

Gets the inbox of a certain actor

```
GET /msg/api/messagebox/{actorId}
```

Description

All the messages will be marked as delivered

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string

Responses

HTTP Code	Description	Schema
200	OK	< InboxMessagePayload > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- message-controller

Sends a message using a certain actor

```
POST /msg/api/send
```

Description

It is necessary to know the Actor Key

Parameters

Type	Name	Description	Schema
Body	**payload** *required*	payload	OutboxMessagePayload

Responses

HTTP Code	Description	Schema
200	OK	SendMessageResponsePayload
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Tags

- message-controller

Gets all log entries for a given message.

```
GET /msg/api/status/{actorId}/{messageId}
```

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string
Path	**messageId** *optional*	ID of message wanted to get log entries for	integer (int64)

Responses

HTTP Code	Description	Schema
200	OK	< Log > array
401	Unauthorized	No Content

HTTP Code	Description	Schema
403	Forbidden	No Content
404	Not Found	No Content

Tags

- message-controller

Gets all actors in own domain, i.e. all potential receivers of internal messages.

```
GET /msg/api/{actorId}/actors
```

Parameters

Type	Name	Description	Schema
Path	**actorId** *optional*	The Actor UUID	string

Responses

HTTP Code	Description	Schema
200	OK	< InternalReceiverPayload > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Tags

- message-controller

Definitions

Swager example Actor

Name	Description	Schema
admins *optional*		< Admin > array
channelRights *optional*		< ChannelRights > array
description *optional*	the actors description	string
domainName *optional*		string
expire *optional*	date until this actor is valid, null for never expire	string
id *optional*	the actors id UUID	string
inbox *optional*		< Delivery > array
name *optional*		string
type *optional*		ActorType

ActorPayload

Name	Schema
channelRights *optional*	< ChannelRightsPayload > array
description *optional*	string

Name	Schema
expire *optional*	string
name *optional*	string
type *optional*	ActorTypePayload

ActorType

Name	Schema
description *optional*	string
id *optional*	string

ActorTypePayload

Name	Schema
description *optional*	string
id *optional*	string

Admin

Name	Schema
domainName *optional*	string
id *optional*	string

ChannelRights

Name	Schema
allowed *optional*	boolean
configSets *optional*	< ConfigSet > array
messageType *optional*	string

ChannelRightsPayload

Name	Schema
allowed *optional*	boolean
configSets *optional*	< ConfigSetPayload > array
messageType *optional*	string

ConfigSet

Name	Schema
attributes *optional*	< string, string > map
id *optional*	integer (int64)
name *optional*	string

ConfigSetPayload

Name	Schema
attributes *optional*	< string, string > map

Name	Schema
id *optional*	integer (int64)
name *optional*	string

CurrentUser

Name	Schema
domain *optional*	string
domainId *optional*	integer (int32)
email *optional*	string
firstname *optional*	string
lastname *optional*	string
maasadmin *optional*	boolean
roles *optional*	< string > array
userId *optional*	string
username *optional*	string

Delivery

Name	Schema
delivered *optional*	boolean
receiverName *optional*	string

Domain

Name	Schema
description *optional*	string
id *optional*	integer (int32)
name *optional*	string

ErrorResponse

Name	Schema
debugMessage *optional*	string
message *optional*	string
status *optional*	enum (100, 101, 102, 103, 200, 201, 202, 203, 204, 205, 206, 207, 208, 226, 300, 301, 302, 303, 304, 305, 307, 308, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 426, 428, 429, 431, 451, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511)
timestamp *optional*	string

InboxMessagePayload

Name	Description	Schema
configId *optional*	ID of config to use for this message. (created in Admin Panel)	integer (int64)
content *required*	Contents of the message	string
delivered *optional*	Determines if the message is already seen or not **Example** : `false`	boolean
meta *optional*	Contains metadata needed for some channels	object
receivers *optional*	Name of Actors to send to for internal messages	< string > array
recipients *optional*	External recipients	< string > array
senderName *optional*	Name of the sender which sent the message	string
subject *optional*		string
type *required*	Message Type	enum (Email, Internal, Slack)

InternalReceiverPayload

Name	Schema
description *optional*	string
name *optional*	string

Iterable« Swager example Actor»

Type : object

Iterable«ActorType»

Type : object

Iterable«ChannelRights»

Type : object

Iterable«CurrentUser»

Type : object

Iterable«Domain»

Type : object

Iterable«InboxMessagePayload»

Type : object

Iterable«MessageType»

Type : object

Log

Name	Schema
description *optional*	string
id *optional*	integer (int64)
messageId *optional*	integer (int64)
messageSubject *optional*	string
time *optional*	string

MessageType

Name	Schema
description *optional*	string
id *optional*	string
properties *optional*	< string, boolean > map

OutboxMessagePayload

Name	Description	Schema
apiKey *optional*	Actor UUID	string
configId *optional*	ID of config to use for this message. (created in Admin Panel)	integer (int64)
content *required*	Contents of the message	string
meta *optional*	Contains metadata needed for some channels	object
receivers *optional*	Name of Actors to send to for internal messages	< string > array
recipients *optional*	External recipients	< string > array
subject *optional*		string
type *required*	Message Type	enum (Email, Internal, Slack)

ReceiverDetails

Name	Schema
actor *optional*	string
code *optional*	integer (int32)
message *optional*	string

ResponseStatusPayload

Name	Schema
code *optional*	integer (int32)
message *optional*	string

SendMessageResponsePayload

Name	Schema
code *optional*	integer (int32)
message *optional*	string
receivers *optional*	< ReceiverDetails > array

F Project Plan

F.1 Goals and boundaries

F.1.1 Background

Headit is developing a lot of different IT solutions for different costumers, when doing that they don't want to make compromises. But messaging systems is often not focused on, due to e.g. lack of resources and economics.

Therefore they want to offer a more general service for message handling to the end user.

F.1.2 Goals

Efficiency targets

- To offer more quality at a cheaper cost: Customers that uses this system will be able to focus more on their core business.
- Efficiency will be improved by removing the previously manual work of informing customers, workers etc. about different events within a company by creating the possibility to automate this process.
- The safety will be increased since it will be possible to track the messages and check if they were sent.
- More control over some business activities in a simple way.

Performance targets

The goal is to create a messaging hub that should work as a stand-alone application and could be configured against external services.

features:

- Store messages tied to application ID, role, or user ID.
- Route the messages through different channels.
- Admin interface for administration and overview of messages
- Possibility to define and send messages through a dashboard (Admin interface)

The objective is to give the customer a better solution that will solve these challenges. They also want to offer better interaction and information flow between customers solutions.

Learning goals

- Get more experience with new technology and tools.
    - Database
    - Java/REST/Frontend (Angular 5)
    - IDE and build tools (Spring Boot)
    - Open Source technology
    - Planning and managing tools (JIRA, Confluence)

- Work experience with complete project from plan to production.
- Cooperate with experienced personnel at Headit.
- Work with “real life” project.

F.1.3 Boundaries

Technical

- The application should be able to run in the customers infrastructure or be delivered as a SaaS.
- Integration could be done through API.
- Admin interface should work on mobile and work with all browsers that support HTML5

Practical

The major difficulty while working on the project could be meetings with the product owner in Hamar. We should be there at least once per two weeks and need to reserve additionally two hours per meeting.

F.2 Scope

F.2.1 Field of study

Along history, we humans have had the need for share with others our thoughts, feelings or what have happened around us. The way we fulfill this need of communication varies depending of the kind of information, to whom this information is transmitted, where is the receiver and the technology that is available at the moment: Sometimes we can just speak directly with another person to transmit our information. In some cases, we need that our information is received by persons that are far away from us physically and/or in time. It can happen that we need that our messages are transmitted immediately. Other times it is important that our message remains unchanged along time.

Over time we have developed many different communication systems: Cave paintings, hieroglyphs, smoke signals, messenger pigeons... And more recently we use phones, e-mails, SMS or social networks to exchange information with other persons.

F.2.2 Delimitation

The exchange of information is not exclusive to human beings. Information is exchanged between machines and between humans and machines and it is very valuable for companies in order to know and serve the needs of its customers in an efficient way. Having access to the right information at the right time and being able to understand it, can a company use its resources effectively and it can constitute a competitive advantage.

In this project, we will develop a system that makes possible to exchange information between the different applications, systems and users that are part of a company. This communication will can be sent using different channels e.g. SMS or email and a copy of this communication must be kept. The system must be able to add new communication channels in the future and in addition it must offer a communication interface so it can be used by many different systems/applications.

F.2.3 Project Description

Headit AS wants to develop a messaging hub (MaaS, Message as a Service) that collects, interprets and reacts to this information. The service will work as stand-alone application and it must be possible to adapt the service to the concrete needs of a every single customer. It should be possible to install it on customers infrastructure or using Cloud technology (SaaS).

The system will:

- Receive and interpret the messages that are received and route them via different channels (mobil app, e-mail, SMS, IoT etc),
- Save the messages linked with the application and person(roll or username) that sent a message.
- Offer a dashboard where the users have an overview over the messages that have been sent before, and where they can define which applications can use the messaging service and which channels will be available for them. It will be possible to send messages to other users using this dashboard.
- Be possible to configure against external services.

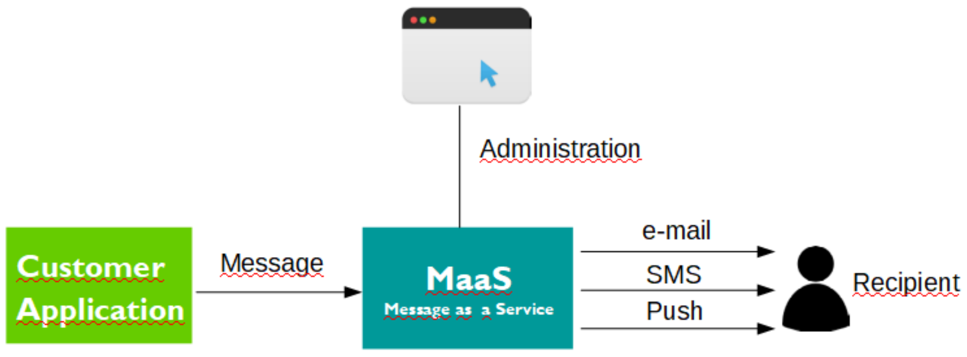


Figure 28: The MaaS (Message as a Service), receives messages from the applications and it forwards them to the final recipient. At the same time a copy fo the messages is kept. The service will have a web application where it will be possible to configure which applications will have permission to use the system

F.3 Project Organisation

F.3.1 Responsibility and Roles

According to Scrum methodology (Choice of methodology is described in appendix F.4.1), we will use the following roles: All group members i.e Manuel Jesús Bravo García, Tom Roar Furunes and Tomasz Rudowski are part of Development team.

Since we are only three and with similar experience in the matter we decided to share a Scrum Master responsibilities by using a combination of “Rotating Scrum Master” and

"Part time scrum master". (https://www.agilealliance.org/glossary/scrum-master)

Each group member will also continue to work as a part of development team (with lower workload) when its his time to take a Scrum Master role. Schedule coming later while working on project timeline.

Stakeholders (Headit) representatives are Bjørn Tore Wiken and Ronny Kristiansen. Ronny Kristiansen will take a role of Product Owner as well.

In addition to roles that arise from chosen methodology we decided to give following responsibilities to group members considering project progress management and reporting:

- Project manager - Tomasz Rudowski
- Contact with stakeholders - Manuel Bravo
- Meeting reports - Manuel Bravo
- New tools - Tom Roar Furunes
- Final report content and correctness - Tomasz Rudowski
- Cloud deployment, Maintenance - Tom Roar Furunes
- Development: Frontend - Tomasz Rudowski
- Development: Backend- Tom Roar Furunes
- Development: libraries for outgoing messages, external systems - Manuel Bravo

F.3.2 Routines and Rules

Following rules has been decided to be used during the project:

- Expected work time is set to 30 hours per week.
- Expected active collaboration and developing new skills necessary to accomplish goal.
- Obligatory meetings with supervisor and stakeholders.
- After each meeting short report is written and added into Meeting Reports log. Written in LaTeX.
- Respect deadline for assigned tasks, inform as soon as possible if not managed to fulfil it.
- When consensus in group cannot be reached, members are obligated to put effort to find a solution satisfactory for all.

Following actions will be taken in case of violation of rules:

- Intern discussion concerning the problem.
- After that formal warning note, with a description of rule violation, written into work-log.
- After the second warning an information send to supervisor.
- Finally formal apply to remove from group. Not applicable within two weeks to project deadline.

F.3.3 Tools

We are going to use several tools to help us trough the project. Here is a list of such tools and their area of application.

- ShareLaTeX - Tool for editing LaTeX-documents. Used for every document created.

- Google drive - Cloud storage. used for storing pictures figures and other files relative to the project
- JIRA - Scrum board/Issue tracker. Used for all events to preform.
- Confluence WiKi. For documentation
- BitBucket. Used for version control (GIT).
- Slack. Used for communication between team members.
- MS Project. Creation of gantt diagrams etc.
- Eclipse IDE for programming in Java.
- Draw.io to make diagrams (Architecture, Class Diagrams etc.)

F.4 Planning, Follow-up, Documentation

F.4.1 Division of the project

Already during the first meeting with Headit we agreed to use a Scrum methodology. Due to the complexity of the project, our experience and the period of time we have, we think that Scrum with sprints of two weeks is an appropriate methodology in order to:

- Better knowledge of the needs of the employer “Headit AS”. At the beginning there will be many unknown requirements and technical specifications that we need to get familiar with. We think that it will be better for us to get known with this requirements as long as we work and get familiar with the system that is going to be implemented.
- Minimize risks: The period of time we have to get done with our bachelor thesis is limited. That’s why we think it’s convenience that we start to deliver some requirements at the beginning of this period of time. It can be risky if we invest too much time specifying requirements that maybe we don’t have time to implement.
- Motivation. We have deadlines every two weeks. We think that this will help us to have control of the development of the project.
- Feedback. We think it will be positive for the development of the project and for us the fact that we will get continuous feedback about of work.
- Learning. In every sprint we will discuss what have we done right and wrong. This will be helpful in order to do every time a better job.

F.4.2 Plan for status meetings and decision points

Project goes over 9 sprints (60 working hours spread over 2 weeks per sprint). We decided on following meetings are important for project progress.

- At the beginning of sprint we will have Sprint Planning Meeting at Hamar
- Each day we will start with intern Daily Meeting to plan tasks for the day and discuss what was done the day before and any difficulty we experienced the previous day.
- Every week we will have meetings with supervisor to confirm project progress
- At the end of the sprint we will have Sprint Review Meeting in Hamar
- After Sprint Review Meeting we will have Sprint Retrospective Meeting to evaluate work done and introduce organisational changes if necessary for the next sprint.

The scrum methodology is flexible and is open to changes, but right now we see the followings decision points:

- 2nd February (Sprint no. 2). Decide basic Architecture to work with.
- 9th February (Sprint no. 3). Decide Design based on accepted Architecture.
- 15th February (Sprint no. 3). Infrastructure and tools must be ready in order to start the implementation.
- 6th April (Sprint no. 6) After this date we will not start to develop new outgoing messaging channels.
- 6th April (Sprint no. 6) Based on current progress we must decide if Mobil App will be implemented or we stay with Web App for PC and focus on another aspects (for example wider choice of outgoing channels).

F.5 Organising and quality assurance

F.5.1 Documentation, standards and source code

We will use the Java code convention for writing Java code [29].

F.5.2 Configuration

We will use Atlassian tools made available to us from Headit. We will use mainly JIRA integrated with Bitbucket and Confluence.

F.5.3 Risk assessment

Below (table 1) we have a table of possible risks, with the probability and effect of this risk. The probability can be low, moderate or high and the effect can be tolerable, serious or catastrophic. The affect tells what part the risk affects; project, product and/or business.

We also have a measurement for some of the worst risks (table 2).

Table 1: Possible incidents

Num	Risk	Affect	Probability	Effect	Measures	Description
1	AWS Not available	Project, Product	Low	Tolerable	No	
2	New channel after deadline	Project	Low	Serious	No	
3	Minor requirement change	Project	High	Tolerable	No	Expected changes in product backlog.
4	Major requirement change	Project	Low	Serious	No	New architecture
5	Change of external systemes	Project, Product	Low	Serious	No	KeyCloak, etc.
6	Tool Change	Project	Moderate	Tolerable	No	IDE, plugin, management system, etc.
7	Programming language change	Project, Product	Low	Catastrophic	Yes	
8	New tech more complicated than expected	Project, Business	Moderate	Serious	Yes	Not enough time to be familiar with new tech
9	Task takes more time than assigned	Project	High	Serious	Yes	Too optimistic planning poker
10	Team member not able to work over long time	Project	Low	Catastrophic	Yes	1/3 of the time gone!
11	Data loss	Project, Product, Business	Low	Catastrophic	Yes	Loss of source code, report, etc.

Table 2: Measurements

Num	Measure
7	To minimise the risk for this happening we will do research to make sure the chosen language is compatible and well suited for the task at hand. If the language still have to be changed we wil make a plan for how this transition is going to happen.
8	To minimise the risk for this happening we have one member of the team assigned to each part of a project who will study this task before everyone else and implementation. We will also use help from stakeholders, supervisor and other experts on NTNU.
9	We need to make sure to communicate that the task takes longer time and if necessary change priorities and/or resources around.
10	We are going to have regular meetings where we inform every team member of what we are working on and how it's working such that everyone have a common understanding of everything. This will minimise the risk for lost work if someone stop working on something.
11	To prevent data loss, we are using cloud services like bitbucket, sharelatex and Google drive to store data. As well as local backups on each team members PC.

F.6 Plan for Implementation

F.6.1 Gantt-diagram

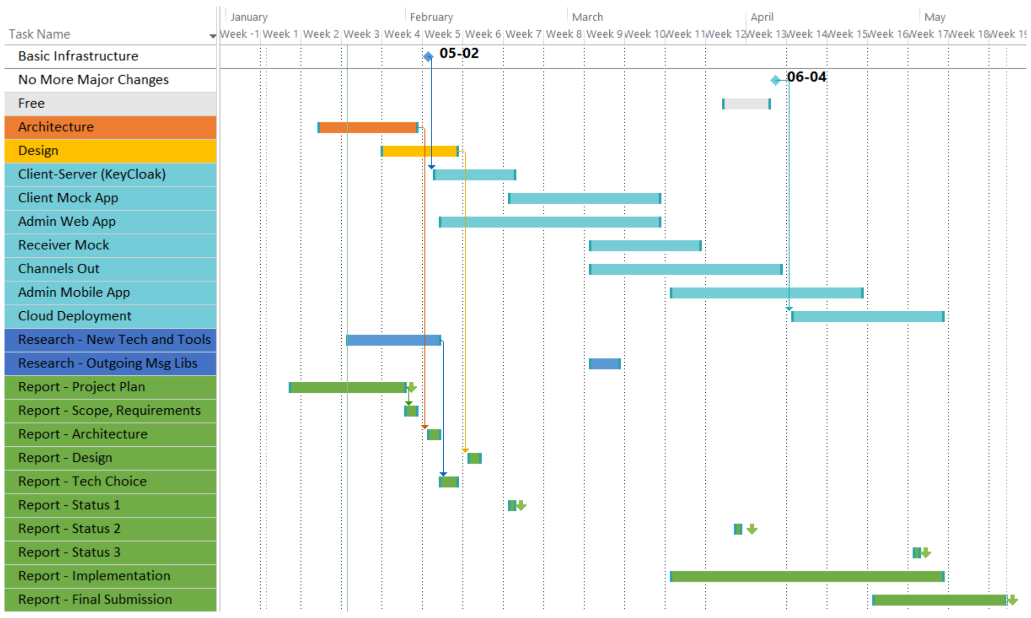


Figure 29: Gantt-diagram with basic activities, MS Project

F.6.2 Milestones and Deadlines

Following deadline have been decided when the project started; all of them concerns report submissions:

- Project Plan - 01.02.2018
- First status report - 20.02.2018
- Second status report - 01.04.2018
- Third status report - 01.05.2018
- Final submission - 16.05.2018

Additionally we have chosen two deadlines to insure completion of both working software and final report.

- Infrastructure - 05.02.2018
- No more major requirement changes - 06.04.2018

The first one is a start of the third sprint when we need to have all necessary tools and resources to start with programming activities. The second one is the end of the sixth sprint, after which we should not accept requirement concerning additional outgoing messages channels in order to reliably deliver the software according to requirements sent until that date; we also consider that this could be a starting point for Cloud Deployment (details will come later in requirements, currently consideration: Docker compose for entire system)

F.6.3 Activity List

Planning

First two sprints are mainly dedicated to create a Project Plan, but after we manage to define and confirm system context, boundaries, interaction and behavioral models we would like to start with next activities i.e. Architecture and Design details.

Architecture

We will create a proposal to an Architecture Design based on initial project description and meetings with stakeholders. Initial Architecture must be created at least at the end of the second sprint, but we are aware and prepared for changes while becoming more familiar with new technology and requirements or external systems change. We would like to employ Architecture Patterns if applicable in defined context.

Design

We will start as soon as significant part of Architecture Design would be confirmed. Since an important part is interaction with external systems we need to focus on details concerning systems confirmed and locked for changes and internal cooperation of main components i.e Message Hub Server - Admin App and Message Hub Server - Client App. We consider use of Design Patterns like for example Adapter to ensure cooperation of components in case of change. We accept that detailed design of parts of the system will need to be updated later.

Programming

This activity may be divided into main groups:

- User authentication. Parallel work on Backend (Server integrated with external authentication system - KeyCloak) and Frontend (Web App for Admin user to interact with server)
- Registration of Client App that may use Message Service. This part may include creating of Mock App instances that may be registered on the Server using Admin App.
- Client App Authentication. Messages send by registered App accepted by Server.
- Implementation of Outgoing Channels. Number and complexity of channels to be decided based on project progress. Must reserve sufficient time for research and integration tasks.
- Mobile Admin App. Potential project enhancement that allows to mobile application instead of desktop version of Admin App.
- Cloud Deployment. Current state of discussion about deployment is that service is supposed to available as "Message as a Service" and technology considered is Docker.

Research

There are many new technologies and tools we need to be familiar with to efficiently work with a project. Most of the time we need to use them at the beginning to support choices made during planning of system design. Additionally it is important to use tools all the way from the beginning; that saves us time for resource migration. Already at this stage we are aware that we need to put an extra effort to familiarise us with state of art concerning Output Channels therefore will reserve time for it at the beginning of the fifth sprint

Report

During the first three sprints most of time dedicated to work with the report will be used to describe planning process and outcomes from requirements, architecture, design and technology choices. We want to work with elements important for the Final Report parallel with implementation, but reserve time during two last sprints to focus on it. Additionally we have decided to reserve some time before each of three partial submission to generate a status report.

F.6.4 Time and Resource Plan

Total time effort to the Project

Based on that each group member will work 30 hours per week, 18 working weeks means we estimate that our effort to the project will be around 1600 working hours. We decided to assign following amounts to main aspects of the project:

- Research and Planning - 300 hours
- Implementation and Deployment - 1000 hours
- Report and Documentation - 300 hours

Sprint plan

We decided to divide work on the project into 9 sprints, each sprint goes over two weeks, which means around 180 working hours each. Dates for all sprints together with main activity planned for each of them are listed below. Sprint no. 6 goes over three weeks, but should have the same amount of working hours considering Easter holidays; we accept that not all engaged parts may be available and active and therefore planned one week off. At least two last days of the last sprint (after final report submission) we need to use for eventual issues concerning infrastructure, accounts, backups, resources.

- Sprint 1 (08.01.2018 - 21.01.2018), Planning, Research
- Sprint 2 (22.01.2018 - 04.02.2018), Architecture, Design, Tools
- Sprint 3 (05.02.2018 - 18.02.2018), Programming
- Sprint 4 (19.02.2018 - 04.03.2018), Programming
- Sprint 5 (05.03.2018 - 18.03.2018), Programming
- Sprint 6 (19.03.2018 - 08.04.2018), Programming
- Sprint 7 (09.04.2018 - 22.04.2018), Deployment, Report
- Sprint 8 (23.04.2018 - 06.05.2018), Report,
- Sprint 9 (07.05.2018 - 20.05.2018), Report, Maintenance

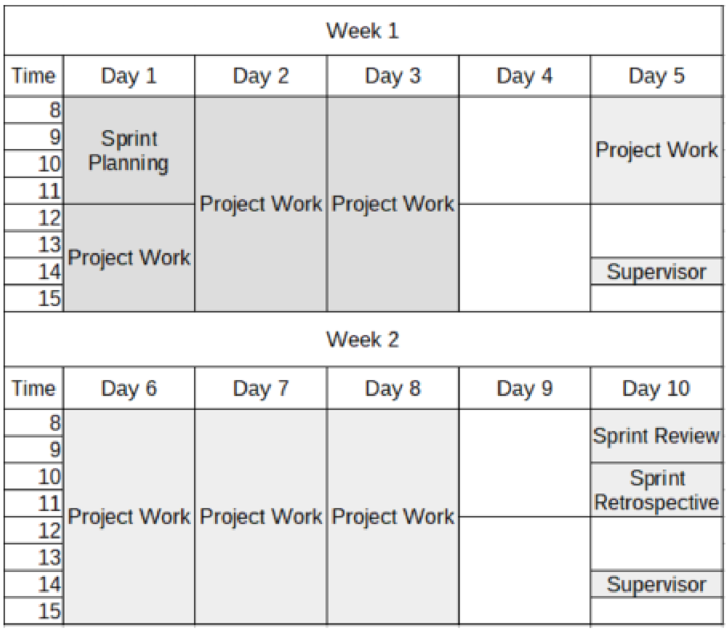
Sprint details

We decided to divide this into three places, where each place contains what we do there.

- Group work at School
    - discuss
    - teach
    - program
    - report overview

- Individual work at home
  - research
  - study
  - program
  - report
- Work at Headit
  - discuss
  - programming
  - work experience.

We have considered following schedule (Picture 30) for each sprint



Week 1					
Time	Day 1	Day 2	Day 3	Day 4	Day 5
8	Sprint Planning	Project Work	Project Work		Project Work
9					
10					
11					
12	Project Work				
13					
14					Supervisor
15					

Week 2					
Time	Day 6	Day 7	Day 8	Day 9	Day 10
8	Project Work	Project Work	Project Work		Sprint Review
9					
10					Sprint Retrospective
11					
12					
13					
14					Supervisor
15					

Figure 30: Schedule for a sprint, time dedicated to the project in grey

G Contract

The contract between Headit and the group members is attached below.

NTNU

Vår dato Vår referanse

Norges teknisk-naturvitenskapelige universitet

Prosjektavtale

mellom NTNU Fakultet for informasjonsteknologi og elektroteknikk (IE) på Gjøvik (utdanningsinstitusjon), og

HEADIT AS

_______________ (oppdragsgiver), og

MANUEL JESÚS BRAVO GARCÍA,

Tom Roar Furunes

TOMASZ RUDOWSKI (student(er))

Avtalen angir avtalepartenes plikter vedrørende gjennomføring av prosjektet og rettigheter til anvendelse av de resultater som prosjektet frembringer:

1. Studenten(e) skal gjennomføre prosjektet i perioden fra 08-01-2018 til 16-05-2018

Studentene skal i denne perioden følge en oppsatt fremdriftsplan der NTNU IE på Gjøvik yter veiledning. Oppdragsgiver yter avtalt prosjektbistand til fastsatte tider. Oppdragsgiver stiller til rådighet kunnskap og materiale som er nødvendig for å få gjennomført prosjektet. Det forutsettes at de gitte problemstillinger det arbeides med er aktuelle og på et nivå tilpasset studentenes faglige kunnskaper. Oppdragsgiver plikter på forespørsel fra NTNU å gi en vurdering av prosjektet vederlagsfritt.

2. Kostnadene ved gjennomføringen av prosjektet dekkes på følgende måte:
   - Oppdragsgiver dekker selv gjennomføring av prosjektet når det gjelder f.eks. materiell, telefon/fax, reiser og nødvendig overnatting på steder langt fra NTNU på Gjøvik. Studentene dekker utgifter for ferdigstillelse av prosjektmateriell.
   - Eiendomsretten til eventuell prototyp tilfaller den som har betalt komponenter og materiell mv. som er brukt til prototypen. Dersom det er nødvendig med større og/eller spesielle investeringer for å få gjennomført prosjektet, må det gjøres en egen avtale mellom partene om eventuell kostnadsfordeling og eiendomsrett.

3. NTNU IE på Gjøvik står ikke som garantist for at det oppdragsgiver har bestilt fungerer etter hensikten, ei heller at prosjektet blir fullført. Prosjektet må anses som en eksamensrelatert oppgave som blir bedømt av intern og ekstern sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.

4. Alle bacheloroppgaver som ikke er klausulert og hvor forfatteren(e) har gitt sitt samtykke til publisering, kan gjøres tilgjengelig via NTNUs institusjonelle arkiv hvis de har skriftlig karakter A, B eller C.

Tilgjengeliggjøring i det åpne arkivet forutsetter avtale om delvis overdragelse av opphavsrett, se «avtale om publisering» (jfr Lov om opphavsrett). Oppdragsgiver og veileder godtar slik offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og instituttleder/fagenhetsleder om de i løpet av prosjektet endrer syn på slik offentliggjøring.

Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode mv. som inngår som del av eller vedlegg til besvarelsen, kan vederlagsfritt benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av NTNU til andre formål, og ikke overlates til utenforstående uten etter avtale med de øvrige parter i denne avtalen. Dette gjelder også firmaer hvor ansatte ved NTNU og/eller studenter har interesser.

5. Besvarelsens spesifikasjoner og resultat kan anvendes i oppdragsgivers egen virksomhet. Gjør studenten(e) i sin besvarelse, eller under arbeidet med den, en patentbar oppfinnelse, gjelder i forholdet mellom oppdragsgiver og student(er) bestemmelsene i Lov om retten til oppfinnelser av 17. april 1970, §§ 4-10.

6. Ut over den offentliggjøring som er nevnt i punkt 4 har studenten(e) ikke rett til å publisere sin besvarelse, det være seg helt eller delvis eller som del i annet arbeide, uten samtykke fra oppdragsgiver. Tilsvarende samtykke må foreligge i forholdet mellom student(er) og faglærer/veileder for det materialet som faglærer/veileder stiller til disposisjon.

7. Studenten(e) leverer oppgavebesvarelsen med vedlegg (pdf) i NTNUs elektroniske eksamenssystem. I tillegg leveres ett eksemplar til oppdragsgiver.

8. Denne avtalen utferdiges med ett eksemplar til hver av partene. På vegne av NTNU, IE er det instituttleder/faggruppeleder som godkjenner avtalen.

9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og NTNU som regulerer nærmere forhold vedrørende bl.a. eiendomsrett, videre bruk, konfidensialitet, kostnadsdekning og økonomisk utnyttelse av resultatene. Dersom oppdragsgiver og student(er) ønsker en videre eller ny avtale med oppdragsgiver, skjer dette uten NTNU som partner.

10. Når NTNU også opptrer som oppdragsgiver, trer NTNU inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.

11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene imellom. Dersom det ikke oppnås enighet, er partene enige om at tvisten løses av voldgift, etter bestemmelsene i tvistemålsloven av 13.8.1915 nr. 6, kapittel 32.

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk

12. Deltakende personer ved prosjektgjennomføringen:

NTNUs veileder (navn):	FRODE HAUG		
Oppdragsgivers kontaktperson (navn):	Ronny Kristiansen		
Student(er) (signatur):	[illegible]	dato	19-01-2018
	Tom Roar Fervnes	dato	19.01-2018
	[illegible]	dato	19.01.2018
		dato	
Oppdragsgiver (signatur):	[illegible]	dato	30.01.2018

Signert avtale leveres digitalt i Blackboard, rom for bacheloroppgaven.
Godkjennes digitalt av instituttleder/faggruppeleder.

Om papirversjon med signatur er ønskelig, må papirversjon leveres til instituttet i tillegg.
Plass for evt sign:

Instituttleder/faggruppeleder (signatur): ______________________ dato __________

H Status Report 1

16.02.2018

H.1 Status for:

- **Planlegging (jfr. fremtidsplan)**: Vi er på riktig spor, ingen bekymringer.
- **Organisering av gruppens arbeid og ansavarsområder:** Foreløpig utfører vi arbeidet som gruppearbeid.
- **Klargjøring av problemstilling:** Vi har en klar idé om de som vi skal utvikle. Endringer har blitt håndtert.
- **Løsningsmetode** Evolusjonær utvikling, rammeverk og teknologi bestemt, jobber med integrasjonsdetaljer.
- **Rapportskriving:** Noterer (log) underveis.

H.2 Totalstatus for punktene over

Vi har en ferdig fremdriftsplan, som vi foreløpig holder oss til. Organisering av gruppens arbeid og ansvarsområder ble definert i prosjektplanen og det har ikke oppstått behov for å endre dem. Når det gjelder problemstillinger har vi hatt flere diskusjoner både sammen og med oppdragsgiveren. Vi har blitt enige med oppdragsgiveren om å utvide omfanget av prosjektet: Den opprinnelige meldingsflyten skal nå også inkludere en intern postboks. Dette har medført noen endringer i konseptuelt design, men vi regner med at vi skal levere alt vi har blitt bedt om innen fristen. Når det gjelder rapportskriving tar vi notater og bilder underveis som skal utnyttes senere.

H.3 Muligheter Trusler/Problemer

Bruk av rammeverk og gjenbrukte komponenter som for eksampel autoriserings-tjeneste (Keycloak) gir oss mulighet å ikke fokusere på sikkerhetsdetaljer og forenkler utvikling, men samtidig forårsaker det en rekke integrasjonsproblemer som vi må vurdere underveis.

H.4 Hva er avsluttet

Vi er ferdige med prosjektplanen, generell arkitektur av systemet (detaljer kan endres underveis da vi blir mer kjent med teknologi som brukes), valg av rammeverk og teknologi vi skal bruke.

H.5 Hva er under arbeid

På grunn av bruk integrasjon av utvalgte teknologi og rammeverk er forsatt under arbeid det klassediagrammet for systemet. Vi jobber nå med første versjon av systemet som inkluderer autoriserings-tjeneste (Keycloak instans i docker) integrert med admin side (server side Spring Boot - Java) og visuell fremvisning (Angular, Bootstrap).

H.6 Tidsfristene

- **Overholdt:** Prosjektplan, Arkitektur, Systemkrav, basis-infrastruktur
- **Overskredet:** Design er ikke ferdig i den grad vi forventet.
- **Kritiske:** Ingen.

H.7 Hva med motivasjon

Vi synes det er motiverende med "daily scrum meetings" der hvor vi bestemmer de ulike oppgavene vi skal jobbe med i løpet av dagen og estimerer tidsbruk. Dagens estimater gir motivasjon til å utføre arbeid i planlagt tid. Gruppearbeid fungerer fint og vi kan støtte hverandre, diskutere og finne løsning.

H.8 Hvordan oppleves veilederkontakt

Vi er fornøyde med veilderkontakten vi hatt til nå.

I Status Report 2

01.04.2018

I.1 Status for:

- **Planlegging (jfr. fremtidsplan)**: Vi er på riktig spor, ingen bekymringer.
- **Organisering av gruppens arbeid og ansavarsområder:** Foreløpig utfører vi arbeidet som gruppearbeid.
- **Klargjøring av problemstilling:** Fokus-endring i problemstilling ble akseptert til utvikling.
- **Løsningsmetode** Evolusjonær utvikling, rammeverk og teknologi bestemt, Tilfredsstillende integrasjonsnivå.
- **Rapportskriving:** Noterer (log) underveis, bilder fra tavle/whiteboard.

I.2 Totalstatus for punktene over

Når det gjelder problemstillinger har vi hatt flere diskusjoner både sammen og med oppdragsgiveren. Vi har blitt enige med oppdragsgiveren om å utvide omfanget av prosjektet: Den opprinnelige meldingsflyten skal nå også inkludere en intern postboks. Dette har medført noen endringer i konseptuelt design. I løpet av tredje sprint implementerte vi intern meldingsboks og den ble akseptert, vi planlegger å jobbe med en bestemt kanal, slik at vi kan sende en kanal ut. Når det gjelder rapportskriving tar vi notater og bilder underveis som skal utnyttes senere.

I.3 Muligheter Trusler/Problemer

Ekstern authoriserings-tjeneste konfigurasjon er fortsatt ikke utnyttet som planlagt. Blir sannsynligvis helt utenfor omfang. Nåværende integrasjonsnivå tilfredsstillende.

Avhengighet av eksterne API som skal motta de meldingene vi sender ut, kan skape ekstra problemer/jobb, for eksempel vi blir nødt til å brke multi-tråd (async task) for å bruke disse kanalene. Problemene ble diskutert med oppdragsgiver.

I.4 Hva er avsluttet

Arkitektur av systemet ble justert i forhold til fokus endring (Interne meldinger). Teknologirammeverk er tatt i bruk og skal ikke endres. Vi har utviklet en tjeneste for å utveksle interne meldinger, tilgjengelig gjennom en web-app. Dette innebærer at databaseløsning er for det meste ferdig.

I.5 Hva er under arbeid

Refaktorering av kode (finpussing, patterns, dokmentasjon) slik at vi får en stabil versjon som er enkel å jobbe videre med. Vi supplerer kildekode med tester. Samler informasjon om første kanal vi skal implementere.

I.6 Tidsfristene

- **Overholdt:** Admin web-app, sender/receiver mock-apps (Erstattet med admin web-app).
- **Overskredet:** Kanal ut ble utsatt pga. fokus-endring (Intern melding).
  I følge prosjektplan skulle vi ha skrevet deler av slutt-rapport (arkitektur/design/scope/tech choice). pga endringer og fokus på implementering, ble dette ikke gjort (Vi skal jobbe med dette i påsken etter refaktorering).
- **Kritiske:** Ingen.

I.7 Hva med motivasjon

Vår motivasjon er den samme som før. Fornøyd med gruppearbeidet.

I.8 Hvordan oppleves veilederkontakt

Vi er fornøyde med veilderkontakten vi hatt til nå.

J Meeting Logs

J.1 12.01.2018 (11:00-11.30) Supervisor, Sprint 1

Frode Haug, Tom Roar Furunes, Tomasz Rudowski
The first meeting with supervisor at NTNU Gjøvik.

It's been mentioned that 3 partial (current work status) reports are required. Dates and content requirements coming later. For now dates are: 20.02, 01.04, 01.05

Discussed key points concerning first submission, i.e. Project Plan. Main suggestions to consider (using paragraph numbers according to pdf from first course meeting):

1.3. consider following aspects:

- practical (where we work, work at campus, etc)
- technical (infrastructure, etc.)

2. most important part, suggested coping to final report – it will become chapter one of final submission

3.2. include also handling of unwanted behavior and ways to force a group member to follow work-flow, rules etc.

4. don't put too much effort to this part

6. Gantt diagram. Finally we need two of them. First created now (planned work-flow) and the second at the end (how work-flow actually looked like). Some of the milestones suggested: Project Plan, 3 status reports. Set deadlines (beslutningspunkter) like for example "we need to decide programming language latest at ..." Proposal: hold timelist in form of xls table for each group member with all dates until project deadline in May. Keep comments short, max 2 sentences to each point to describe what was done. Proposal: regular meetings on Fridays at 11:00 - to consider change of time if we are on meetings in Hamar

J.2 19.01.2018 (09:00-11.00). Headit, Sprint 1

Bjørn Tore Wiken, Ronny Kristiansen, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski
Today we had the first meeting with Headit. The goal of this meeting was to share with them our perception of the system, ask some questions and show them the planning we have made.

Following points were clarified:

- **Automation and efficiency:** The purpose of the messaging system is to help Headit's customers to make automatic and improve the efficiency of their business systems. However this is out of the scope of the system we are going to develop. Our focus will be a system that receives messages and forwards them through different channels.
- **Interface:** Headit made us concerned about that the messaging service is intended to be used by many different systems. It was discussed the need of an API.

- **Interface, functionality:** The MaaS must be able to tell to a system which channels are available for this system.
- **Design, new features:** It's important that the design facilitates that new features can be added in the future. E.g. To give feedback about a message. (If it was received or not etc).
- **Design, new channels:** It must be possible and easy to add new channels in the future.
- **Design, several users:** The messaging service must be able to work with several users at the same time.
- **KeyCloack:** Headit thinks that KeyCloak can be out of scope.
- **Log:** The system must be able to save every message that is send through it.
- Logg meldingene???
- **Privacy law:** Before this system is put in production, it is necessary to accomplish the law requirements that apply. However Headit says that we will not work on it right now. Of course this must be mentioned in the documentation.
- **Requirement, limit of messages** Begrensing av antall meldinger som kan sendes.
- **Requirement, Security:** It its discussed that we can make use of HTTPS in order to take care of the confidentiality of the messages. However Headit proposes that we can start working with HTTP at a first stage and so change into HTTPS when we already know that some functionality of the system works.
- **Dictionary:** Headit asked us to make a dictionary where we describe the different parts of our system. E.g. Maas (Message as a Service).
- **Design, Communication Channels:**Probably it will not be possible to implement all kind of communication channel during the semester. Headit tells us that we can focus in those channels we are more interested in.
- **? Access Rights to send messages** Will it be by app-level or by user-level.
- **Database, storage:** Headit asks for an openSource technology that is widely used.
- **Database, Encryption:** If it is not implemented it should be taken care in the design.
- **User requirement, superAdmin:** The system will have a super administrator with right to read all the messages.
- **User Requirement, Key for the applications:** The Maas will make the key for an application on demand. The user can get this key by "copy-paste" or by SMS.
- **User Requirement, new usuers:** It must be possible to add new users and add a new superAdmin for every customer.
- **User Requirement, UserID:** Assume that an user has the same ID in every application of a customer.
- **User requirement, System messages:** It must be possible to send messages to "just" one APP.
- **Coding:** Standard format
- **Developing, Admin App:**Ha focus one Admin Web. Obs! med mobilAPP. Er det hensiktsmessige?

J.3 19.01.2018 (14:00-14.30). Supervisor, Sprint 1

Frode Haug, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski
We met our supervisor and we reported him about our first meeting we had with Headit.

So, we reviewed and we got feedback about the draft of the Project Plan we delivered some days ago. Supervisor pointed out some mistakes and gave us some comments.

J.4 24.01.2018 Headit, Sprint 2

Bjørn Tore Wiken, Ronny Kristiansen, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

J.4.1 Sprint Review of Sprint 1

We reviewed the work we did during the first sprint:

- **Requirements:** We have impression that the list of requirements is OK. However we still have not validate it.
- **Project Plan:** The project plan is done.
- **Technology Research:** We have researched the followings technologies:
  - Spring (Websites through Java):
  - Keycloack (Open Source Identity Access Mananger): Right now difficult. Headit told us during the meeting that probably it might not be used)
- **Architecture:** We have a already a good understanding of the main components that will be part of the architecture: REST API, DB User, DB App, DB MsgLog, Adapters for the outgoing messaging channels and a authorisation component.
- **Tool Setup:** Headit provided us an account in Jira, Confluence and BitBucket. In Confluence we have worked with list of requirements. In Jira we haven't set up a backlog yet and we need to connect it with list of requirements. Repository on Bitbucket is alredy set up, but still is not integrated with the other tools.

J.4.2 Sprint Planning of Sprint 2

1. **(!) Validate Requirements.**
2. **(3) Create a System Specification Document**.
3. **(!) Project Plan:** Validate Project Plan with supervisor (We will have a meeting with him on Friday the 26th january).
4. **(2) Spring:**All group memebers should run a Hello World example.
5. **(2) KeyCloack:** We have to wait for a more detailed technology specification of the system, but we can get more familiar with it.
6. **(3) IDE:** We have not taken a decision yet about the IDE we will use. We need to confirm if we are going to use Eclipse or another IDE:
7. **Confluence:** (3)Summaries of the meetings with Headit must be added . (!)We have to close the initial Requirement List. (3)We are not sure if we will create a detail diagramas for all the requirements.
8. **(!, Depends on requirements validation) JIRA:** Create Backlog linked to Confluence.
9. **(3) BitBucket:**Init repo on all local machines and run a test.
10. **(1, Depends on Validate Requirements) Architecture Design:** Research current state of the art solutions. Break Components into smaller pieces. Research Architectural patterns. Create Architectural document
11. **(1, Depends on Arquitecture Design) Design and Implementation:** Class Diagram. DB Model. User Interface (Admin Panel, API). Research Design Patterns.

Research libs and state of the art technology. State Diagram, Sequence Diagram (Class perspective).

After we did the spring meeting, we started to work with Architecture Design. We drew a schema with of the messaging system and its modules. We identified as well some possible patterns, that we can use.

- Facade
- MVC
- Broker

We must check if these patterns can be a good choice for us.

We need to clarify with Headit the roles that will have access to the system and their functions.

J.5 26.01.2018 (14.15-14.30) Supervisor, Sprint 2

Frode Haug, Tom Roar Furunes, Tomasz Rudowski, Manuel Bravo

Reviewed the last changes we did to the document.

J.6 31.01.2018 (12-14) Headit, Sprint 2

Bjørn Tore Wiken, Ronny Kristiansen, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

Additional meeting during Sprint 2.

MaasAdmin: can control own apps inside own domain, one user account many roles (e.g: MaasAdmin og AppAdmin).

MaasAdmin: will be able to configure the applications of all the AppAdmins and in all domains. MaaSadmin is the only one who is allowed to create new domains.

MaasAdmin: MaasAdmin has access to the message log (all the messages). However the AppAdmin can encrypt the messages of its own domain. (Out of Scope).

Maas: It is expected that MaaS can diferentiate users of applications that use the MaaS.

AppAdmin: It must be registered in Maas. AppAdmin are managed in the internal KeyCloack. Appadmin creates keys for applications.

Extra Functionality (?) Registering: We discussed that it could be desirable to import users from an external keykloack in order to change the identification model. (From app-based to user based)

Functionality: Maas Admin can send a link to the new AppAdmin in order to register in the Maas.

Functionality: Maas must have an interface where the App admins can create their access keys for apps.

Functionality: Apps will have access to all or just certain channels. It will be the AppAdmin who decides it.

PUSH: No constraints about the way that PUSH messages will be sent. We will find the best solution (Url, Webhook, Google...)

Messages: There are two kind of messages (internal and external). Both will have a timestamp. It must be possible to know who has received/read an internal message. Every message will be logged on every step. E.g When it was received by the Maas, and when it was sent by a "channel". It is desirable to track the messages in order to discover possible fails. As well is desired to logg all the activites that take place in the Maas (e.g. inlogging, change in App configurations etc.) Be aware that scope can be adjusted. (Log in a file or in a DB).

Functionality: It must be possible how many messages are sent on a channel. (The just mentioned SMS). Anyway this is already registered in the log.

Functionality: The same API for request/send messages.

MVC: Considering Admin panel GUI, use MVC modell based on Spring Web, angular and BootStrap.

OpenSource: It is desirable with open source components that are well known and widely used to make easy its maintain. Eventual license issues will be solved later.

Efficiency Goal: It was mentioned the following goals: To offer more quality at a lower cost. Possibility to focus on the core business instead of support systems like messaging. Security and assurance that messages were sent. Tracking possibilities.

It is expected that the system will work as cloud service. So it will be nice if we could deliver it with docker compose file.

Architecture: It was discussed use App based indentification with keys that are created by the AppAdmin.

New features: We must adjust the system in order that it will be possible to add new channels in the future.

Database: We will decide which kind of DB we will use

J.7 07.02.2018 (12-14) Headit, Sprint 3

Bjørn Tore Wiken, Ronny Kristiansen, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

We presented our solutions to the requirements from last meeting and it was accepted. We presented a Database E-R Model. Probably more details will be added lately. (Sql, noSQL not decided yet).

Following points were decided during the meeting

- Consider Messages as blog entry if using NoSQL. (See Doc).
- Documentation of correct use of MaaS system. End user documentation.
- Possibility of decorate messages with tags/flags in order to use expire dates in different situations / different ways. (e.g. Pop up message om å change password).
- MaaS should not interprete the content of the messages
- Possibility to use access filters to outgoing channels. (Apps, users,..)
- Possibility to send a message to a group of users. Details of this must be described in system specification for end users.
- Maas will support delivery confirmation but no read confirmation (Apps IDE must take care of this if relevant)
- We can make use of Keycloack's settings and stylesheets of our admin page.
- Extend posssible recipients users, roles, groups

- We presented our proposal of backlog. Headit's comment are: Some epics are too big, consider epic as a complete solution for one problem. Current epics looks like components. Examples of epics are: User Configuration, Message Box solution. Some User Stories are too big. Consider make them epics.
- Use a wider perspective of the system. Even if we don't implement them, we can point them. However we will focus on essential functionality. It should be possible to show the routing of messaging in order to present the value of the software in a complete form.
- The original Keycloack panel should be allowed to just MaaS Admin.
- Consider use of Spring Security.
- Scope: If we don't manage to make Keycloack work as a authrorization service in a short time, we should skip it and focus on functionality that lies behind KeyCload like MaaS Admin and message handler.
- Nice to have is not a requirement.
- Order issues list in backlog by priority.
- Estimate task in relation to each other (Task 1 twice time as task 2). User story points in the beginning or get experience in future sprints.
- We will make a sprint backlog for the next spring during the Spring Planning Meeting. However the product owner will be able to accept it or change it.

J.8 21.02.2018 (9-11.15) Headit, Sprint 4

Bjørn Tore Wiken, Ronny Kristiansen, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

The KeyCloak services integration we manage until now are enough. We could possibly use it wider but do not put too much attention to this (out of scope). Alternative solution we presented is accepted. i.e admin panel secured with keycloak login, message api secured with intern check against token list (db)

Advanced use of keycloak settings (realms, groups, roles) can be dealt with later, important to focus on key functionality, i.e message forwarding.

Database solution (MariaDB) is accepted. SQL config/setup file should run automatically when docker started.

Separate Frontend and Backend into two projects. Run them parallel, eventually in docker as separate services.

Advantages

- No session ID when changing from admin panel to API
- Many potential customers will have separate projects

Disadvantages

- Code refactoring (time?)

UUID we create for each sender are accepted, but should be known only by owner, i.e app knows only its own id, admin knows his own id and all ids of his apps. We should create some unique abstraction names that could be used as "Receiver" in all messages instead of unknown UUID. Add e.g dropdown list of all available receivers not using

UUID but abstract names

Temporary we have only Admin Panel, all MaaS Admin activity is done in Keycloak Admin UI. It is ok for now.

Spring Security vs Keycloak-Spring: do not use too much time on changing security. the one we have now works ok.

Deleting of tokens (UUID). Do not delete, mark inactive/expired e.g flag or expiry date column in a table.

J.9 07.03.2018 (09-11) Headit, Sprint 5

Bjørn Tore Wiken, Ronny Kristiansen, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

J.9.1 Sprint Review of Sprint 4

We reviewed the work we did during the second sprint, following comments were made:

- Docker works fine in the way we implemented it
- Application.properties works fine to have all in one file
- rename Receiver (InternalReceiver ?) so it is more obvious that it can be null for external messages only
- Consider removing foreign key from Log (separate tables ?) How to get message content then?
- Proposal to keep only internal messages in database (SMS, email only in Log) How about tracking back external messages when not possible to store message content (no message content in Log)? We store all accepted messages until further decided.
- Log format accepted (include also list of actors involved). Do not use complex tree structure use simple "text logfile" format. Use filters.
- Long term policy for data storage (out of scope)
- Adjust MessageController message so message is accepted if at least one receiver is found.
- Response from message API (200, array of status objects + response message). Status object is code (e.g 200, 400) and receiverId (ActorId or unrecognized string)
- Consider global Error handling (see Sprint libraries)
- Headit got access two repositories we are using for back- and frontend
- Frontend code: keep Angular components small with possibility to move them (UI redesign), but do not put effort into at that could be done by users.
- OK to keep message content as a string (JSON payload parsed to string - the string stored in database and send to converter - the converter extracts JSON from string and finds neccesary meta data there (if fails it will be stored as an error log entry)

J.9.2 Sprint Planning of Sprint 5

In this Sprint we should focus on processing messages through more stages rather then covering all API on first stage.

- Focus on Internal messages (send between registered Actors)
- Architecture change! Internal messages are stored in database when they arrive and are accepted. It is no point to send them to converter and forward to another

InboxDB. Message table has functionality of an InboxDB allready, messages could be picked from there.
- Model adjustment (Delivered/Read flag in Message table)
- Model adjustment. Add Subject field to Message class (and request payload) null for e.g SMS should be handled with default values ('no-subject')
- Internal message Log (1 - received, 2 - delivered)
- External message Log (1 - received, 2 - converted, 3 - sendToExternalProvider, 4 - responseFromExternalProvider)
- Accepted proposal of using Sonar for code quality, but it is better to focus on own tests to check functionality.
- Set timezone in Docker (time-sync problem)
- Tests! Important to write good tests for system functionality. This could help with refactoring, code quality.
- Model adjustment. Access rights for Actors (channel based, which channels are accessible)
- Model adjustment. Channel config (individual config of meta-data for a pair Actor-Channel, possible multiply entries for each Actor-Channel too choose)

J.10 21.03.2018 (09-11) Headit, Sprint 6

Bjørn Tore Wiken, Ronny Kristiansen, Tom Roar Furunes, Tomasz Rudowski

J.10.1 Sprint Review of Sprint 5

- Demo different access rights (Actor by Channel)
- Demo different config sets (by Actor and Channel)
- Demo Messagebox tab in Admin Panel, accepted
- Public Config in a form we proposed accepted, to adjust use only config.id, config.name on a client side, supply Message when accepted with fields store in a config (by Id)
- discussed project plan (gannt diagram), ok
- discussed changes in GUI, do not focus on edit/save Actor in Details view, current proposal works fine as a demo for future
- no need for complex multithread programing, evt. Async task can be added later if outgoing channel need longtime task

J.10.2 Sprint Planning of Sprint 6

- add timestamp to Delivery and show it in Log beside Delivered flag
- different API ? for client and admin so it should not be marked as delivered if admin views inbox that belongs to an app
- API for message status (by msg.id) for async task, but possible to use it in other situations
- Focus on create one complete route for message (chosen proposal - Slack) - Converter - Facade - ChannelOut (Log on every step)
- proposal/concept use Slack ChannelOut as an additional warning channel set by admin on Events (e.g. msg rejected)
- tests should be run against functionality not API directly.

- Consider architecture change on a backend. Controller as small as possible - Service with Business Logic (here goes tests) ->Repositories ->DB, or Controller ->Service ->Adapter/Facade (for forwarding messages)
- tests should be integral part of all components, required (?) to set task as DONE
- do not test trivial functionality like getter/setter
- research: integrate DB in IntelliJ and use design from there instead of phpMyAdmin

J.11 05.04.2018, Supervisor, Sprint 6

Frode Haug, Tom Roar Furunes, Tomasz Rudowski

Meeting with supervisor. Rapport questions.

- Chapter 1. Introduction. Goal of the project (why we chose it ?) from our perspective.
- Report target is a censor, but use language at student level
- Treat end-user requirement as Headit requirements; and as following application (not sorce code) as a product. Could discuss codebase requirements later in report?
- Do not describe why technology chosen when we use technology name for the first time. Use "more about why we chose this" and discuss it later in eg. Discussion chapter part Alternatives/Choices or Conclusion chapter.
- Scope for used technology description: new for us after five semesters can be starting perspective. Angular, TypeScript, Spring, JPA, Keycloak, Docker (practical, Slack + webhook. (?)Jira, (?)Confluence.
- Diagrams. If many that looks similar put them in appendix and use one of them as an example in text.
- More focus on report writing. Agreed about partial report submission every week from now.

J.12 11.04.2018 (09-11) Headit, Sprint 7

Bjørn Tore Wiken, Ronny Kristiansen, Tom Roar Furunes, Tomasz Rudowski

J.12.1 Sprint Review of Sprint 6

We reviewed the work we did during previous sprint, following comments were made:

- Discussed refactored code, accepted.
- Presented all tasks, concerning functionality, planned for sprint - accepted with some change proposals we agreed to adjust, see planning of next sprint
- Presented unit tests - ok, focus on main functionality (use 75% coverage as a base but comment if decided to have less). Mockito.
- Discussed wider use of Keycloak API, decided to keep user config (MaaS Admin role) directly through Keycloak admin site.

J.12.2 Sprint Planning of Sprint 7

- We presented alternative solutions, technology choices etc. We agreed that they should not be implemented but would be nice if we describe them. - Place it in the report.
- Discussed outgoing channel "email" and proposed two solutions; first internal email

server; second use of external provider and store credentials in the properties file. - we should go for the second alternative, use external email account for demo purposes.
- Functionality adjustment. Set priority of message data, like eg. Receiver. If set in custom message it should overwrite metadata in payload, lowest priority to config set. But this is not final, make decisions for each channel, and document priority policy.
- Documentation. How to setup system in development/production environment.
- Documentation. Expand documentation for what channels are available, and how to use them (payload doc concerning end users)
- Documentation. Explain functionality from MaaS Admin perspective. (how to add users etc)
- Functionality adjustment. Consider that API calls that result in errors could return documentation, which describe how to use API.
- If we have time could check Swagger for documentation instead of Confluence.
- New functionality. Introduce Domain for an Actor. Receiver of message should be only from within Domain. Check when message receive by API, choice of receiver from admin panel now expanded to all Actors from Domain.
- Functionality adjustment. OK to keep Actor types (user) and (app) but no need to setup an (user) Actor access key for each admin as default. Limit access to (user) type Actor messagebox to admin panel only (not allowed through open /msg API)

J.13 13.04.2018, Supervisor, Sprint 7

Frode Haug, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

Report feedback.

J.14 20.04.2018, Supervisor, Sprint 7

Frode Haug, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

Report feedback.

J.15 25.04.2018 (09-11) Headit, Sprint 8

Bjørn Tore Wiken, Ronny Kristiansen, Tom Roar Furunes, Tomasz Rudowski

J.15.1 Sprint Review of Sprint 7

- Approved solution for domains.
- Approved e-mail channel solution.
- Approved Swagger documentation.

J.15.2 Sprint Planning of Sprint 8

- Present API errors in nice way at Admin Panel.
- Improve Swagger documentation
- Control code base for bugs, unclear documentation etc, but also enhancement pos-

sibilities.

J.16 03.05.2018, Supervisor, Sprint 8

Frode Haug, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

Report feedback.

J.17 09.05.2018 (09-11) Headit, Sprint 9

Bjørn Tore Wiken, Ronny Kristiansen, Manuel Bravo, Tom Roar Furunes, Tomasz Rudowski

Presented last bug fixes and refactored alert service on frontend. Discussed presentation time/content.

K Worklog

Below the worklog is presented, status on the date:
15.05.2018

	Tom		Manuel		Tomasz		Tom	Manuel	Tomasz
Date	**Hours**	**Note**	**Hours**	**Note**	**Hours**	**Note**			
12.01.2018	2,5	Group meeting, Supervisor meeting	2	Group meeting	2,5	Group meeting, Supervisor meeting			
13.01.2018	1	Created this worklog and other documents	1	Forprosjekt. Field of Study					
14.01.2018					1	Project plan, Organisation	3,5	3	3,5
15.01.2018	7	Group meeting, Project plan	7,5	Group meeting, Project Plan	8,5	Group meeting, Project Plan			
16.01.2018	6,5	Group meeting, Project plan, latex formating	4,5	Group meeting,	4,5	Group meeting			
17.01.2018	6	Project plan, Study	6	Project plan, Scenario	7	Project Plan, Scheduling, System Design			
18.01.2018	3,5	Group meeting	3,5	Group meeting	3,5	Group meeting			
19.01.2018	6	SPM, Supervisor meeting, group meeting	6	SPM, Supervisor meeting, group meeting	6	SPM, Supervisor meeting, group meeting			
20.01.2018									
21.01.2018							29	27,5	29,5
22.01.2018	8,5	Group meeting	8,5	Group meeting	8,5	Group meeting			
23.01.2018	4	Requirements, Project plan, Study	3	Meeting reports, Requirements.	5	requirements, created Confluence page			
24.01.2018	4	Group meeting	4	Group meeting	4,5	Group meeting			
25.01.2018									
26.01.2018	4,5	Group meeting, supervisor meeting	4,5	Group meeting, supervisor meeting	4,5	Group meeting, supervisor meeting			
27.01.2018									
28.01.2018	1,5	Keycloak/Spring	2,5		2,5	Architecture patterns	22,5	22,5	25
29.01.2018	9	Group meeting - patterns	9	Group meeting - patterns	9	Group meeting - patterns			
30.01.2018	4,5	Research sping+angular+boostrap			4,5	Architecture patterns, FCM			
31.01.2018	4	SPM	4	SPM	4	SPM			
01.02.2018									
02.02.2018	4,5	GM (meeting review, epics), SM	4,5	GM (meeting review, epics), SM	4,5	GM (meeting review, epics), SM			
03.02.2018									
04.02.2018	4	Spring+keycloak+angular	4	Db, Spring og angular	2	db-model, xampp	26	21,5	24
05.02.2018	7,5	GM	7,5	GM	8,5	GM, draw AD, DBM			
06.02.2018	5	db, angular, docker	7,5	Spring, Angular, Db	9	GM, db, Angular, docker			
07.02.2018	7	GM, SPM	7	GM, SPM	7	GM, SPM			
08.02.2018					7,5	Angular, Spring, Auth			
09.02.2018	9,5	GM, angular, spring, kc	4	GM	4,5	GM,			
10.02.2018	4,5	Spring, kc	9	KeyCloack, Angular					
11.02.2018	1	angular			2	Docker, Repo	34,5	35	38,5
12.02.2018	8	GM, angular	8	Gm, Angular	8,5	GM, keycloak			
13.02.2018	11	GM, Angular Frontend	8	Gm, Spring, Backend	8	Gm, Backend			
14.02.2018	8	GM, Frontend	8	GM, Frontend	8	GM, Backend			
15.02.2018									
16.02.2018	2	sr1	2	sr1	2	sr1			
17.02.2018					1	db			
18.02.2018					2	docker, test	29	26	29,5
19.02.2018	8	GM, frontend	7	GM, frontend	7	GM, db, api			
20.02.2018	8	GM, frontend	8	GM, frontend	8	GM, msg auth			
21.02.2018	5	SPM	5	SPM	5	SPM			
22.02.2018									
23.02.2018	4	GM	4	GM	4	GM			
24.02.2018	5	docker			1,5	DB			
25.02.2018	1	db (study)			6,5	DB	31	24	32
26.02.2018	9	GM, DB	9,5	GM, DB	9,5	GM, DB			
27.02.2018	8	GM, Frontend, jpa	8,5	GM, Actors API, JPA	8,5	GM, Actors API, Angular			
28.02.2018	8,5	GM, Frontend,	8,5	GM, Frontend, Backend	8,5	GM, Backend			
01.03.2018									
02.03.2018	4	GM, API spec	4	GM, API spec	4	GM, API spec			
03.03.2018	6	Frontend			1,5	API spec			
04.03.2018	5	Backend, Frontend	5	Backend, Frontend			40,5	35,5	32
05.03.2018	8	Backend	5	Frontend	8,5	Backend			
06.03.2018	7	Review, Backend, Frontend	7	Review, Backend, Frontend	5	review, demo			
07.03.2018	5	Hamar, review	5	Hamar, review	5	Hamar, review			
08.03.2018									
09.03.2018	3,5	merge v1.0			3,5	merge v1.0			
10.03.2018	5	frontend, docker							
11.03.2018							28,5	17	22
12.03.2018	9	frontend, msgbox	8,5		9				
13.03.2018	6	frontend, msgbox	8	frontend, msgbox	8	frontend, msgbox			
14.03.2018	6,5	gm, fontend, backend	6,5	gm, fontend, backend	6,5	gm, fontend, backend			
15.03.2018									
16.03.2018	10	unit tests	8,5	Gm, Frontend	10,5	actor config			
17.03.2018									
18.03.2018							31,5	31,5	34
19.03.2018	11	GM, unit tests, frontend	11	GM, Actor Config	10,5	GM, Actor config			
20.03.2018			8	Gm, Actor Config	8	GM, Actor config			
21.03.2018	4	Hamar, raport			4	Hamar, raport			
22.03.2018	4	GM	4	GM	4	GM			
23.03.2018	9	GM, backend refactor	6,5	GM, exeptions	4	GM			
24.03.2018									
25.03.2018	3	backend refactor	5	Exceptions			31	34,5	30,5
26.03.2018	6,5	GM	9	GM	11	GM, raport			
27.03.2018	7	GM, Code Review	7	Gm, Code review	7	GM, Code review			
28.03.2018									
29.03.2018									
30.03.2018									
31.03.2018									
01.04.2018							13,5	16	18
02.04.2018									
03.04.2018									
04.04.2018	6	Slack			7	Slack			
05.04.2018	6	Raport	6	Raport	6	Report			
06.04.2018	2	GM	2	GM	2	GM			
07.04.2018	3	send message with config							
08.04.2018							17	8	15

Message Service

	Tom		Manuel		Tomasz		Tom	Manuel	Tomasz	
Date	Hours	Note	Hours	Note	Hours	Note				
09.04.2018	8,5	API Spec	8,5	Backend	8,5	report				
10.04.2018	8	API Spec, Javadoc	8	Javadoc	8	report				
11.04.2018	9	SPM, GM	9	SPM, GM	9	SPM, GM				
12.04.2018										
13.04.2018	9	GM, SVM. Docker Research	7	GM, SVM	7	GM, SVM				
14.04.2018	4	docker								
15.04.2018	4,5	backend					43	32,5	32,5	
16.04.2018			3,5	Kommentering / Rapport						
17.04.2018	8,5	Report	8,5	Report	8,5	report				
18.04.2018	9,5	Report, GRATULERER MED DAGEN!!!	9,5	Report	9,5	Report				
19.04.2018	7,5	Report	7,5	Report	7,5	Report				
20.04.2018	7,5		7,5		7,5					
21.04.2018	1	api doc								
22.04.2018	2	Email channel			9,5	Domain	36	36,5	42,5	
23.04.2018										
24.04.2018	7	Report	7	Report	5	report				
25.04.2018	10	SPM, Report	10	SPM, Report	10	SPM, Report				
26.04.2018	10	Report	10	Report	10	Report				
27.04.2018			6	report, SM	6	report, SM				
28.04.2018										
29.04.2018			5	Alert Service			27	38	31	
30.04.2018										
01.05.2018	7,5	report	7,5	report	7,5	report				
02.05.2018			3	alert	5	report				
03.05.2018										
04.05.2018	7	report	7	report	7	report				
05.05.2018										
06.05.2018	6	Report, alert			4	report	20,5	17,5	23,5	
07.05.2018	7	Report, Frontend	4,5	Report	8,5	Report				
08.05.2018	7	Report	7	Report	7	Report				
09.05.2018	3	Spm	3	SPM	3	SPM				
10.05.2018	7,5	Gm	7,5	GM	7,5	Gm				
11.05.2018	6	Frontend, Backend	6	Frontend, Backend	6	Backend tester				
12.05.2018										
13.05.2018							30,5	28	32	
14.05.2018	8,5	Report, Frontend, Backend	8,5	Report, Frontend, Backend	8,5	Report				
15.05.2018	8	Report, Frontend, Backend	8	Report, Frontend, Backend	8	Report				6
16.05.2018							16,5	16,5	16,5	
							511	471	511,5	
							1493,5	TOTAL		