



Norwegian University of  
Science and Technology

# MGiNX - Creating a modern platform for managing email delivery

Author(s)

Tobias Lønnerød Madsen  
Ernst Thomas Kvadsheim Sem-Jacobsen

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

18.05.2016

Supervisor(s)

Frode Haug

## Sammendrag av Bacheloroppgaven

Tittel:	<b>MGiNX - Creating a modern platform for managing email delivery</b>
Dato:	18.05.2016
Deltakere:	Tobias Lønnerød Madsen Ernst Thomas Kvadsheim Sem-Jacobsen
Veiledere:	Frode Haug
Oppdragsgiver:	Norwegian University of Science and Technology
Kontaktperson:	Consignor AS v/ Øystein Ranvik & Bjørn E. Pedersen
Nøkkelord:	ASP.NET, Postgres, Epost, Tilbakemelding, IMT
Antall sider:	127
Antall vedlegg:	16
Tilgjengelighet:	Åpen

---

Sammendrag:	Denne oppgaven omhandler utviklingen av en epost-motor, som har ansvaret for å informere mottakere av pakker om oppdateringer, samt å hente inn tilbakemelding fra mottakere. Disse epostene og tilbakemeldingskjemaene kan enkelt justeres av aktøren, som også har mulighet til å endre grafisk profil, bruke flere språk basert på pakke mottaker med mer. Oppgaven dekker alle aspekter fra design og implementasjon, til distribusjon og testing.
-------------	--

## Summary of Graduate Project

Title:	<b>MGiNX - Creating a modern platform for managing email delivery</b>
Date:	18.05.2016
Authors:	Tobias Lønnerød Madsen Ernst Thomas Kvadsheim Sem-Jacobsen
Supervisor:	Frode Haug
Employer:	Norwegian University of Science and Technology
Contact Person:	Consignor AS v/ Øystein Ranvik & Bjørn E. Pedersen
Keywords:	ASP.NET, Postgres, Email, Survey, IMT
Pages:	<a href="#">127</a>
Attachments:	16
Availability:	Open

---

**Abstract:** This thesis is about creating an email engine for sending shipment updates to receivers using a user-defined email configuration with optional survey delivery. The email configuration should be easily adjusted by the actor, and provide tools for managing languages and branding options. The thesis will describe all parts of the system, from design to implementation to deployment and testing.

## Preface

### Meet the team



Tobias



Thomas

First of we would like to give a big thanks to Consignor for providing us with the opportunity to work on this project. In addition we thank Bjørn E. Pedersen and Øystein Ranvik who represented Consignor during the project, and have been very forthcoming with us.

We would also like to thank our supervisor Frode Haug who have been with us on this journey from start to finish. With weekly meetings Frode has always been available if we had any problems, and even if we had nothing important to say it was still nice to take a break from work and have a chat.

Finally I, Thomas, want to give a special thanks to Tobias for allowing me to be a part of this project and sticking with me to the very end. This has been an amazing experience, and Tobias has been an awesome partner who I've learned a lot from over the last few months we've spent together.

# Contents

<b>Preface</b> . . . . .	<b>iii</b>
<b>Contents</b> . . . . .	<b>iv</b>
<b>List of Figures</b> . . . . .	<b>vii</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
1.1 Introduction . . . . .	1
1.1.1 Field of study . . . . .	1
1.1.2 Technical Scope . . . . .	1
1.1.3 Project Description . . . . .	2
1.2 Project Restrictions . . . . .	3
1.3 Target Audience . . . . .	4
1.3.1 Application Audience . . . . .	4
1.3.2 Report Audience . . . . .	5
1.4 Purpose . . . . .	5
1.4.1 Why this assignment . . . . .	5
1.4.2 Learning Objectives . . . . .	5
1.4.3 Impact Objectives . . . . .	6
1.4.4 Performance Objectives . . . . .	6
1.5 Academic Background . . . . .	6
1.6 Roles . . . . .	6
1.7 Report Structure . . . . .	7
<b>2 Requirements</b> . . . . .	<b>9</b>
2.1 Functional Requirements . . . . .	9
2.1.1 Use Cases . . . . .	9
2.2 Interface With Consignor . . . . .	13
2.3 Supplementary Requirements . . . . .	15
2.3.1 Deployment Platform . . . . .	15
2.3.2 Usability . . . . .	15
2.3.3 Compatibility . . . . .	15
2.3.4 Language support . . . . .	16
2.3.5 Security & Authentication . . . . .	16
2.3.6 Documentation and testing . . . . .	16
2.3.7 Logging . . . . .	17
2.4 Product Backlog . . . . .	17
<b>3 Design</b> . . . . .	<b>18</b>
3.1 System Architecture . . . . .	18

3.2	Sending Trustworthy Emails . . . . .	19
3.2.1	Reverse DNS . . . . .	20
3.2.2	SPF . . . . .	20
3.2.3	DKIM . . . . .	20
3.2.4	Encryption . . . . .	21
3.3	Wireframes and frontend planning . . . . .	21
3.3.1	Wireframes . . . . .	22
3.3.2	Graphical style . . . . .	24
3.4	File structure . . . . .	25
3.4.1	Application Organization . . . . .	25
3.4.2	User Data . . . . .	26
<b>4</b>	<b>Implementation . . . . .</b>	<b>27</b>
4.1	Tools & Codebase . . . . .	27
4.2	ASP.Net Core . . . . .	27
4.2.1	Entity 7 ORM . . . . .	28
4.2.2	Release Candidate troubles . . . . .	28
4.3	Database . . . . .	29
4.4	Frontend . . . . .	30
4.4.1	Intro . . . . .	30
4.4.2	Dashboard . . . . .	31
4.4.3	Reports . . . . .	32
4.4.4	Email Editor . . . . .	32
4.4.5	Survey Editor . . . . .	37
4.4.6	Branding . . . . .	38
4.4.7	Settings . . . . .	39
4.4.8	Survey . . . . .	39
4.5	Backend . . . . .	40
4.5.1	API Endpoints . . . . .	40
4.5.2	Email Engine . . . . .	41
<b>5</b>	<b>Testing and Quality Assurance . . . . .</b>	<b>44</b>
5.1	Unit Testing . . . . .	44
5.1.1	Unit test example . . . . .	45
5.2	Component & System Testing . . . . .	46
5.3	Acceptance Testing . . . . .	47
5.3.1	Compliance Testing . . . . .	48
5.4	Swagger . . . . .	48
<b>6</b>	<b>Deployment . . . . .</b>	<b>50</b>
6.1	Docker . . . . .	50
6.1.1	Postgres . . . . .	50
6.1.2	nginx Proxy . . . . .	51

6.1.3	nginx Let's Encrypt Companion . . . . .	51
6.1.4	MGiNX . . . . .	51
6.1.5	Swagger UI . . . . .	53
6.2	Encryption . . . . .	53
6.3	Postfix . . . . .	53
<b>7</b>	<b>Conclusion . . . . .</b>	<b>55</b>
7.1	Results . . . . .	55
7.2	What would we do differently today? . . . . .	55
7.3	Further Development . . . . .	56
7.4	Group Evaluation . . . . .	56
7.5	Conclusion . . . . .	57
	<b>Bibliography . . . . .</b>	<b>58</b>
<b>A</b>	<b>Terminology . . . . .</b>	<b>61</b>
<b>B</b>	<b>Project Description . . . . .</b>	<b>63</b>
<b>C</b>	<b>Project Agreement . . . . .</b>	<b>66</b>
<b>D</b>	<b>Group Rules . . . . .</b>	<b>69</b>
<b>E</b>	<b>Hour Log . . . . .</b>	<b>71</b>
<b>F</b>	<b>Project Plan . . . . .</b>	<b>75</b>
<b>G</b>	<b>Meeting Summaries . . . . .</b>	<b>91</b>
<b>H</b>	<b>Status Reports . . . . .</b>	<b>95</b>
<b>I</b>	<b>Email Reader Compatability Tests . . . . .</b>	<b>100</b>
<b>J</b>	<b>NuGet Configuration . . . . .</b>	<b>104</b>
<b>K</b>	<b>Bower Configuration . . . . .</b>	<b>107</b>
<b>L</b>	<b>UserController.cs . . . . .</b>	<b>109</b>
<b>M</b>	<b>UserRepository.cs . . . . .</b>	<b>113</b>
<b>N</b>	<b>Sprint Log . . . . .</b>	<b>117</b>
<b>O</b>	<b>Swagger UI . . . . .</b>	<b>124</b>
<b>P</b>	<b>Project Poster . . . . .</b>	<b>126</b>

## List of Figures

1	Scrum Roles. . . . .	7
2	Use Case Diagram. . . . .	10
3	Sequence Diagram. . . . .	14
4	Diagram of system architecture. . . . .	18
5	Domain model diagram. . . . .	19
6	Wireframe of the landing page. . . . .	22
7	Wireframe of the dashboard. . . . .	22
8	Wireframe of the email editor. . . . .	23
9	Wireframe of the survey editor. . . . .	24
10	Wireframe of the survey page. . . . .	24
11	Font and color representation. . . . .	25
12	Diagram showing file structure of application. . . . .	25
13	.Net Architecture Diagram. . . . .	28
14	Screenshot of application dashboard. . . . .	31
15	Screenshot of report. . . . .	32
16	Screenshot of email editor. . . . .	33
17	Screenshot of email editor with toolbar and content. . . . .	34
18	Screenshots of email editor with and without expanders. . . . .	35
19	Screenshot of variable dialog. . . . .	37
20	Screenshot of the survey editor. . . . .	37
21	Screenshot of branding options. . . . .	38
22	Screenshot of application desttings. . . . .	39
23	Screenshot of survey page. . . . .	40
24	Screenshot of testing tool for emulating shipments. . . . .	47
25	Swagger UI Documentation. . . . .	49
26	Diagram of host topology. . . . .	50



# 1 Introduction

## 1.1 Introduction

Consignor (previously EDI-Soft) is an IT-company headquartered in Oslo who offer solutions that aim to support large and small companies in their delivery management making it easier, less time consuming and cheaper for them to deliver packages to their customers. This solution integrates with the client's existing online ordering system and assist in printing crucial shipment documents, providing customers a vast carrier library containing both national and international carriers, calculating and comparing shipment costs for chosen carriers, alerting customers and carriers of on-going and completed shipment events as well as tracking the status of shipments.

### 1.1.1 Field of study

Due to the ease of online shopping many people choose to go online to satisfy their consumer needs, as anything can be bought and delivered straight to your home with just the press of a button. Retailers have acknowledged that the user experience is a crucial part of the transaction, hence a lot of resources have been spent to improve the online shopping experience, and there exists a lot of good solutions here today. However, during the transaction there is a disconnect between retailer and customer. For retailers the transaction is completed once the carrier collects the package, but for the customer the shipment phase takes up the majority of the transaction. During this phase there are a lot of factors that impact the overall customer satisfaction that the retailer has little to no control over once the package has been dispatched, and some they might even be unaware of. This link between the retailer and customer is one of the challenges we will attempt to provide a solution for with this project.

### 1.1.2 Technical Scope

As previously mentioned Consignor specializes in developing delivery management software, with one of the components, the email editor, being somewhat underdeveloped. The email editor is used to define the contents of emails sent to package receivers through simple text boxes, and optional checkboxes which adds additional shipment specific data. This solution is easy to use, but suffers from limited functionality. It currently doesn't offer any capabilities in terms of altering the layout or design of generated emails, resulting in purely text based emails. This means users are unable to customize the service to suit their setting, and as a consequence of this the customer experience appears impersonal. Their current solution also lacks any sort of sender receiver communication channel, therefore offering very little insight into how their delivery management decisions impact their customers.

Over the course of this project we will develop an email editor that builds on the ideas of their existing editor, as well as expanding on it to reflect the current market needs. This will be implemented as a web application using modern web development practices.

A back-end Web API using RESTful practices will also be developed to handle communication with the web application, storing user data in addition to handling the creation and relaying of emails.

### 1.1.3 Project Description

The application we have been tasked by Consignor to develop will consist of four main components.

1. **Email Editor:** Used to easily customize email content and layout
2. **Email Engine:** Used to inflate email configurations with relevant information
3. **Survey Editor:** Used to configure surveys to be sent to receivers
4. **Report:** Used to display statistics and survey results

The following section will go into more detail on each of these components.

#### **Email Editor**

The email editor will provide users with predefined templates which all offer a unique layout and design which can be used as is, or modified to better suit the individual customers needs. Users with a more technical background will also be able to create these templates from scratch by feeding the engine premade HTML configurations, as well as tweaking the HTML of existing templates. Using a non-technical interface the editor should be easy to use, and allow users to customize templates by adding or removing text fields, images, links, attachments and tokens. Tokens are placeholder elements that represent variables specific to each shipment, and can therefore not be added as static elements. Instead, these will be populated with appropriate data by the email engine, some examples of tokens are estimated arrival date, receiver name and receiver address. Finally, the user can change the color palette for each template to better blend with the companies branding.

Emails will be sent on a per event basis. The editor is therefore split into a set of shipment events. The events available to the sender will be defined by Consignor at a later stage, but for the purpose of this project we have been given three examples we can work with: shipment sent, arrived and picked up. Receivers require different information at each event, so an email configuration will be saved for each event. The system will also keep track of several languages, with one configuration per language. When adding a new language you should at all times be able to copy the text, images and layout from a different language and event configuration.

#### **Email Engine**

The email engine should be implemented as a webservice, and is notified by Consignors system when a shipment has reached certain events. Based on the event an email will be generated using a customized configuration based on the actor, event and language. The engine will take care of all details in creating the email, including adding static or dynamic attachments, inflating tokens with data, and logging system data. The email engine should send emails on behalf of the user, and as such the system will need to get the required configuration from Consignor. To prevent the server in question from being blacklisted as spam, the system will occasionally have to check DNS records, to see if the

required SPF (Sender Policy Framework) records are up to date and allows the system to send emails on behalf of the user. If the SPF records are not found, an error message should appear. We may also have to fall back to using an in-house sender email to ensure service availability.

### **Survey Editor**

The editor will allow users to add up to 10 questions for each configuration, this limit is added to ensure that surveys do not appear overwhelming. Furthermore, questions will primarily be using a rating system from 1 to 5, however a text option should also be available for users who believe a more detailed answer is fitting. Similar to emails, surveys will also keep track of configurations on a per language basis.

Consignor takes the approach that receivers are generally not very fond of taking part in surveys, and therefore wants surveys to be quick and easy to complete. To accomplish this surveys should be made using responsive web design, with a simplistic layout.

### **Report**

The report will provide users with statistics over the average survey rating and total surveys answered each month. There should also be a list of all survey answers, which users can order by location, carrier, language and so on, making it considerably easier to identify outliers in their delivery management. Ultimately this should function as a continuous feedback loop for users, alerting them of what they can improve on in the future, in addition to knowing which carriers perform better in what area.

## **1.2 Project Restrictions**

Project restrictions manifested themselves primarily in two ways, through project management as well as design and implementation decisions made during development. In the next section we will discuss our project management process and some of the restrictions that stem from our choices, as well as technical restrictions.

### **Project Management**

When looking at what development methodology to follow it was quickly revealed that an agile approach was more suitable than a plan driven one. This was due to time restrictions, the need for a working prototype at the end of the semester and most importantly the agile willingness, and ability, to accommodate for changing requirements. This was important, as even though the core functionality of our project is set, uncertainty lies in how these are developed in addition to requirements beyond Consignor's initial vision.

Scrum was chosen due to its focus on a structured work-flow, as well as Consignor's previous experience with Scrum in the workplace, a process Tobias has previously been a part of. By following the Scrum methodology every sprint will result in a working increment of the final product which will be used during meetings with both supervisor and Consignor to provide a status report. This will benefit the team greatly as it gives a more accurate representation of how the team is performing. This, however, caused problems when we started writing the bachelor report. The report is structured with a plan-driven process in mind, forcing us to adapt our experience with Scrum to suit the given report structure.

For our configuration of Scrum we will be following a two week sprint cycle. Workdays will be from Monday to Friday at NTNU in Gjøvik from 10:00 to 17:00, the exception being Fridays when the workday will start at 12:00 due to additional classes.

Every sprint will end in a meeting with Consignor, combining both the sprint planning and sprint review meeting. During these meetings we will provide Consignor with a short demo of the current iteration of the project. This gives them a clear understanding of the current status of the project, as well as an opportunity to give continuous feedback on the direction they want to go moving forward. The rest of the meeting is used to plan what should be included in the next iteration, and hence what should be developed over the course of the next sprint.

From Consignor we were given our own room, including the option to work at their headquarters in Oslo for the duration of the bachelor project. Even though this would give us a permanent workplace we decided to decline the offer, as the time lost from the daily commute would outweigh the benefits.

We planned to meet with the supervisor every week on Tuesdays at 13:30 to review the project status and make sure the team is working at a steady pace so as to not fall behind. As we get further into development meetings with the supervisor became less frequent as the team got more organized, and the need for weekly status reports was reduced.

### **Technical Restrictions**

1. Our system is unaware of any data related to package transactions, and will therefore rely on Consignor's existing system to provide updates to a package shipment status.
2. Our system is to provide a webpage in which receivers can provide feedback on their experience with the sender and transactions between them. This feedback page will be provided through e-mail, and as per Consignor's request, this webpage is required to be responsive as to accommodate both desktop and mobile devices.
3. We will not be doing any work to integrate with Consignors existing software. However, we will be using sample data and examples of configurations provided by Consignor for testing purposes to make the system as compliant as possible, for an easier integration at a later point.
4. The front-end will be developed using HTML5 and Twitter Bootstrap, and will therefore aim at supporting newer versions of Chrome, Firefox, Internet Explorer, Safari and Microsoft Edge.

## **1.3 Target Audience**

### **1.3.1 Application Audience**

The application has two separate audiences, one for the receiver-oriented email and survey components, and another for the dashboard, email and survey editor components. Consignor is not limited to Norwegian customers, making the product aimed at an international audience.

## **Email & Survey**

The consumers for these components are shipment receivers. They are non-technical end-users, and often on small mobile devices. As such, the UI needs to be intuitive and responsive.

## **Dashboard, Email Editor & Survey editor**

The consumers for these components are shipment senders and are bound to desktop computers. They range from light users with a non-technical skillset, to power users with a background in computers who can fully utilize the more complex features of the editors.

### **1.3.2 Report Audience**

The report primarily targets the examiner and our supervisor, but also fellow students and developers who can use this report as a learning tool for their own bachelor project, as well as personal projects. The report will give readers insight into how we developed the final product, detailing our work process, design choices as well as implementation and deployment of the finished product. The report assumes the reader has at least a basic understanding of computer science, and is written with this in mind. The report is written in English, this felt the most natural to the team due to the fact that a lot of technical words do not have a Norwegian counterpart, and the ones that do, often sound bizarre. This should help the report appear more fluid, as well as making it available to a broader audience.

## **1.4 Purpose**

### **1.4.1 Why this assignment**

With Tobias already having done an internship-like summer with Consignor, Tobias arranged for Consignor to write this project proposal so that he would get relevant experience in the development environment of choice at Consignor. The project proposal is very interesting in the rapidly expanding field of informatics in logistics, so getting Thomas on board was no problem. The project also lets us make a lot of our own choices for implementing and fulfilling the requirements, and involves on a lot of different technologies, allowing us to involve ourselves in the entire stack. The reasoning behind developing from scratch instead of adapting existing products was because Consignor required a great deal of flexibility and customization for actors in the software, as well as being able to accept Consignor-generated shipment data. If we were to adapt existing products, the end result would be messy and too loosely coupled.

Prior to development we were given the task by Consignor to name the software we were going to develop. The name we ultimately stuck with was MGiNX, short for Mail engine X, which sadly got slightly confusing later in development once we started using nginx [1] in our project.

### **1.4.2 Learning Objectives**

For this project, we aim to gain knowledge in a number of fields and technologies. Some of them being:

1. ASP.Net Core Technologies using WebAPI

2. Web technologies for creating responsive applications
3. Scrum Software Methodology
4. System Architecture using RESTful practices and API driven web architecture
5. Project planning and execution

### **1.4.3 Impact Objectives**

Provide Consignor with a proof of concept software that can be easily integrated with their existing system, and is designed to provide clients of Consignor with a system in which they can customize their user experience to suit their needs.

### **1.4.4 Performance Objectives**

Considering the growing market of logistics software and a need for adaptability for users, we aim to fulfill the following performance objectives:

1. Create a easily extensible and maintainable system
2. Create a scalable and robust system
3. Give more control to senders surrounding branding and communications with receivers of shipments
4. Collect and display relevant statistics regarding communications with receivers and receiver happiness and satisfaction

## **1.5 Academic Background**

Over the last three years both team members have studied a bachelor in computer science at NTNU in Gjøvik. This has given us a breath of knowledge within a multitude of languages and technologies, such as C++, Java, SQL databases, scripting as well as plan-based and agile methodologies and practices. Through optional courses we each have our unique and valuable experiences and ideas. From courses like Object-Oriented Software Development, Thomas has gotten a more thorough understanding of how and when architecture and design patterns should be used. In addition, the course provided an in-depth view and analysis of available development tools and their uses, providing a great toolbox for the bachelor project. Especially useful for Tobias was a Human Computer Interaction course, which focused on creating user friendly experiences throughout entire applications. Both team members have experience making websites and web applications, using a plethora of frameworks and libraries such as knockout.js, JQuery, Foundation 5 and Bootstrap just to mention a few.

## **1.6 Roles**

Early in the planning phase we divided roles and responsibilities into two separate categories, the once covered by Scrum and those specific to us as bachelor students. The following graphic shows how these roles were delegated.

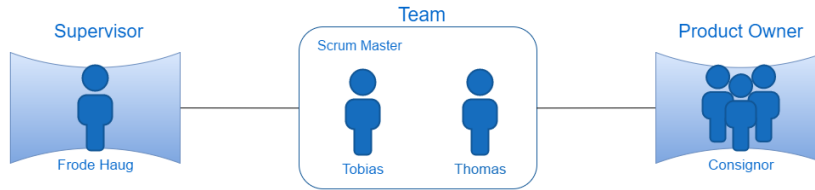


Figure 1: Scrum Roles.

The role of team leader and Scrum master was given to Tobias, since he had worked at Consignor over the course of the previous summer. This opportunity gave him insight into Consignors work-flow, and more specifically how their configuration of Scrum worked. Tobias' weekly responsibilities involved scheduling meetings with Consignor, writing meeting summaries and managing the weekly docker builds. As Scrum Master he also took on some of the responsibilities of the product owner, such as managing the product backlog. This was done due to our desire to be more self organizing, and not rely too heavily on Consignor during the individual sprints.

Thomas acted as a standard team member, with a few additional administrative tasks. This included booking weekly rooms at NTNU in Gjøvik to ensure the team always had a private workplace so as to not be disturbed during work hours, managing the weekly hour log in addition to setting up and maintaining the back-end and the testing framework.

Bjørn E. Pedersen and Øystein Ranvik will be representing Consignor over the course of this project, expressing their needs and wishes for the final product. As such they will act as product owners for the duration of this project, and as per our contract are obligated to be available for consultancy on a bi-weekly basis. They have, however, gone far beyond their contractual obligation, offering to pay for any licenses as well as travels costs that we deemed necessary for the completion of the project.

The responsibilities of our supervisor, Frode Haug, fall somewhat outside of the traditional roles of Scrum, his role was more of a supportive one. He was there to ensure the team maintained a realistic vision for the final product, in addition to not deviating from their sprint goals through weekly meetings.

## 1.7 Report Structure

**Introduction** Provides the reader with an overview over the project and the team members, as well as what our purpose is with this assignment.

**Requirements** This chapter lists both the functional and supplementary requirements present in this project. The relation between our system and Consignors existing system is also explained.

**Design** Describes the early design stage of the process, with visual and technical ideas and plans.

**Implementation** Highlights some of the technology we used, and showcases many of the

components of the application.

**Testing & Quality Assurance** This chapter goes through the different testing methods used throughout development, as well as other methods used to assure a certain level of quality.

**Deployment** Gives a brief overview of what is necessary for server deployment of the application.

**Conclusion** Summarizes the report and project, along with some discussion about the results.



## 2 Requirements

### 2.1 Functional Requirements

From the project description provided by Consignor (see appendix B) we made a use case diagram to clarify and illustrate the activities and actors present in our project. With the use case diagram as a basis we created a set of use cases, each detailing the functionality within an element of the diagram. These were presented to Consignor and used to establish the initial functional requirements of the system, later refined and added to the first iteration of the product backlog.

Through the use of scrum the list of requirements expanded with every sprint, adding new requirements as well as fleshing out old ones. This process meant that the complete set of requirements agreed upon towards the end of the final sprint was different from the initial scope. This was natural as Scrum allows for many small iterations ultimately culminating in a final product, as apposed to a plan-based approach where all requirements are established during the planning phase. The following chapter is an accumulation of all requirements established throughout the course of this project. A visualization of the use cases can be seen in diagram 2.

#### 2.1.1 Use Cases

Even though we are using an agile approach with the Scrum framework, we feel it's more fitting for this kind of project to use a more use-case oriented structure rather than user stories. Our use cases do however resemble the user story format, with a short description and minimal other requirements. This creates a more thorough and richer use case, describing both the needs of the participating actor, in addition to detailing the steps needed to complete it.

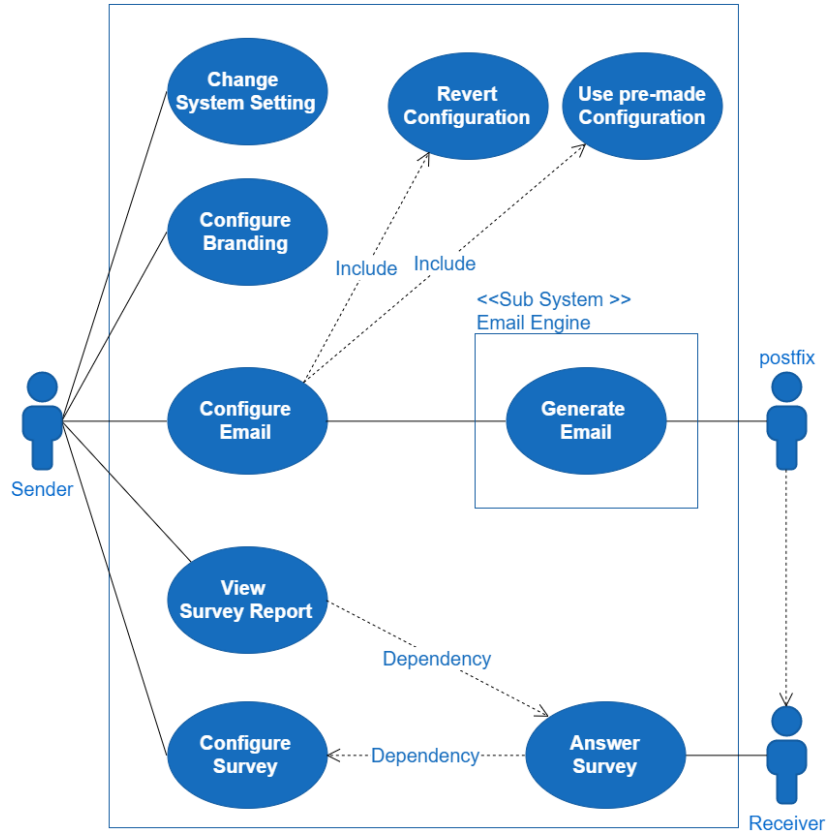


Figure 2: Use Case Diagram of MGiNX.

### Use case [RUC1]: Answer survey

**Description:** As a receiver I want a web interface in which I can give feedback and rate the service provided to me by the sender, accessible through both desktop and mobile devices.

**Priority:** 8

**Pre Conditions:**

- A survey must have been configured by sender
- The receiver must have received the package

**Standard flow:**

1. Receiver receives a link to the survey through email or SMS
2. Receiver clicks the link and is redirected to the survey website
3. Receiver fills out the survey and clicks send

## Use case [suc1]: Generate email

**Description:** As a sender I want the system to automatically generate emails to receivers at certain events.

**Priority:** 10

**Pre Conditions:**

- A configuration for the specific email event must have been defined

**Standard flow:**

1. The system receives a notification signaling that a specific shipment has reached an event
2. The system generates an email based on the configuration and branding of the specific actor
3. The email is sent to the receiver

## Use case [suc2-1]: Configure email

**Description:** As a Sender I would like to change the content of a email configuration for a given event in a given language.

**Priority:** 10

**Pre Conditions:**

- SUC7-1 must have been done

**Standard flow:**

1. User clicks 'Configure Email' from dashboard
2. User selects event
3. User selects language OR remains on default language configuration
4. User edits email configuration with editor
  - User can edit or add text and hyperlinks
  - User can format text
  - User can insert variables
  - User can insert or edit images
5. User clicks save

## Use case [suc2-2]: Use pre-made Configuration

**Description:** As a Sender I would like to have the system use a pre-made email configuration.

**Priority:** 6

**Pre Conditions:**

- SUC7-1 must have been done

**Standard flow:**

1. User clicks 'Configure Email' from dashboard
2. User selects event
3. User selects language OR remains on default language configuration
4. User click 'Switch to HTML view'
5. User inserts pre-made configuration
6. User clicks save

## Use case [suc2-3]: Revert Configuration

**Description:** As a Sender I would like to revert to a previous email configuration.

**Priority:** 5

**Pre Conditions:**

- SUC7-1 must have been done

**Standard flow:**

1. User clicks 'Configure Email' from dashboard
2. User selects event
3. User selects language OR remains on default language configuration
4. User picks iteration for the 'Previous Iterations' column
5. User clicks save

## Use case [suc3-1]: Configure survey

**Description:** As a Sender I would like to create and edit surveys that is to be sent to users based on language.

**Priority:** 10

**Pre Conditions:**

- SUC7-1 must have been done

**Standard flow:**

1. User clicks 'Configure Survey' from dashboard
2. User selects language OR remains on default language configuration
3. User deletes or add questions to survey
4. User edits question title and rating method
5. User clicks save

### **Use case [suc5-1]: View Survey reports**

**Description:** As a Sender I would like to view reports describing survey results

**Priority:** 9

**Pre Conditions:**

- SUC7-1 must have been done

### **Use case [suc6-1]: Configure branding**

**Description:** As a Sender I want to be able to add my companies branding, so that it can be used in emails to the receiver as well as the survey.

**Priority:** 4 **Standard flow:**

1. User clicks configure branding
2. user adds a primary and secondary color scheme
3. User adds the company logo
4. User clicks save

### **Use case [suc7-1]: System settings**

**Description:** As a Sender I would like to be able to set the sender email address of outgoing email, as well as be able to set the default email language for all events.

**Priority:** 10

**Pre Conditions:**

- DNS have to be configured correctly

**Standard flow:**

1. User clicks 'Configure System' from dashboard OR during first-time setup
2. User changes default language from a drop-down list of all registered languages
3. User changes sender email address in the appropriate field
4. System checks DNS records
5. User clicks save

## **2.2 Interface With Consignor**

The final product should be able to function both as part of Consignors environment, as well as a stand-alone system. This meant that having a clear view of how these systems were separated in addition to where they intertwined was crucial to success. To achieve this we made a sequence diagram illustrating the operations required to create and send an email. When a shipment event occurs our system is notified of this through a JSON package sent by Consignor, containing all shipment data needed to send an email to the correct receiver. When MGiNX receives the shipment data, our system will generate an email, as illustrated in the diagram below.

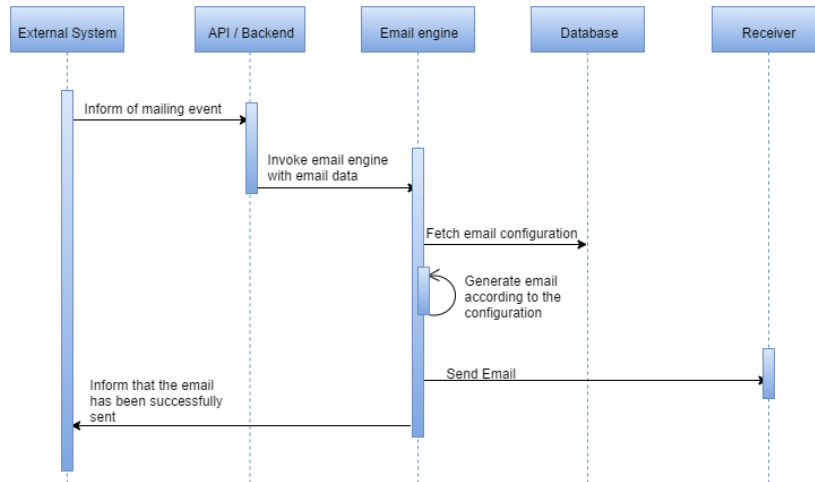


Figure 3: Sequence Diagram illustrating sending of an email.

In this diagram, "External System" refers to the part of Consignors system that will interact with our application. For testing purpose we made a mock up of the external system to be able to send these packages to our system our self. How this was done will be discussed in detail in chapter 5: Testing and QA on page 44.

During a planning meeting with Consignor, we were given an example JSON payload, to get a general idea of what data are available, and how the format of the payloads are structured. A simplified version of the payload looked like this:

```

1 {
2   "InstallationID": "000",
3   "ShpNo": "000",
4   "OrderNo": "000",
5   "Addresses": [{
6     "Kind": 1,
7     "Name": "...",
8     "Street": "...",
9     "PostCode": "0000",
10    "City": "...",
11    "Email": "..@..",
12    "CountryCode": "no",
13  }]
14 }
  
```

Based on this, our email compiler will parse and navigate the payload based on what is needed from the email configuration defined for the specific actor, event and 'CountryCode'. Some of the attribute keys will be simplified in the editor, and we will implement a mapping from easy token names to JSON paths. As an example of this, one of the editor tokens might be named "ReceiverName" and mapped to the JSON path "Addresses[0].Name" when navigating the JSON payload.

## 2.3 Supplementary Requirements

### 2.3.1 Deployment Platform

A philosophy we followed closely over the course of this project was that the final product should have as few limitations as possible. This included only using freely licensed libraries, such as MIT and Apache allowing commercial use, an ORM for database operations and finally not being restricted to a single deployment platform. Historically ASP.Net has been limited to running solely on Windows servers, but with the release of ASP.Net Core RC1 came the promise of multi-platform support, allowing application to run on Windows, Linux and OSX servers [2].

### 2.3.2 Usability

Having been tasked with developing an application partially inspired by Consignors existing system in which usability was largely lacking made this an important aspect if we were to provide any value to Consignor. Software developers used to primarily focus on the functionality offered by the application, with design and usability being more of an afterthought, if even considered at all. Nowadays, however, the longevity of any application is heavily rooted in its ease of use, in conjunction with the functionality that is offered. IBM even as early as in 1981 were aware of this fact, and to this day view usability with equal importance to functionality [3]. Firstly though we need to define the term usability, as it can be somewhat vague, and by itself does not offer a lot of value. Jakob Nielsen divides the term into five components, three of which we will be looking at: learnability, efficiency and errors. [4]. Learnability refers to how easy it is for users to accomplish basic tasks the first time they encounter the design. Efficiency refers to once users have learned the design how quickly can they perform more complicated tasks. Errors refer to how many errors do users make, how severe are these errors, and how easily can they recover from user errors?

To address these requirements we used a two part design philosophy. Firstly we wanted the base functionality to be simplistic and easy to grasp, and secondly offer more complex tools that if used efficiently provide much greater control over the end result. How we accomplished this is covered in more detail on chapter 4: Implementation on page 27. To ensure no data is lost through user error, any changes made to emails and surveys can be reverted at any time granted the user hasn't saved changes made to the original configuration. However, even if the user does end up saving a malformed configuration, we will be keeping a list of the previous four iterations, allowing users to load these into the editor as the current configuration.

### 2.3.3 Compatibility

Compatibility is a major factor when you want your application to be available to as many people as possible, and our application is no exception. For our project there are two compatibility issues we will have to address, email readers and browsers.

#### Browser Compatibility

We were given few requirements when it came to browser compatibility, as long as it works with most modern browsers. Based on statistical data [5] and software availability, we set a requirement to support at least the following browsers:

- Chrome
- Internet Explorer 10+
- Firefox

### **Email Reader Compatibility**

As the most important compatibility target, email reader compatibility is a crucial requirement for the project. This way we can ensure the templates we make will work no matter what. The target clients for testing is primarily the most popular desktop and web email readers [6], with older Microsoft Outlook clients being one of the toughest targets to comply with. We ended up thoroughly testing the following clients:

- Microsoft Outlook 2013
- Microsoft Outlook 2003
- Google Gmail
- Yahoo! Mail
- Mozilla Thunderbird

### **2.3.4 Language support**

Consignor operates in Norway, Sweden, Denmark, Finland, China and the United Kingdom, providing Consignor with an international audience. As the target audience is split into two groups, sender and receiver, there's an important distinction to make in terms of what part of the software should be localized for each region.

For this project we will assume that the senders are well versed in English, and will therefore not spend any resources towards internationalization of the final product. With the receivers, however, we can not make such an assumption. Instead, we will leave the translation up to the individual sender, by providing them with the option to add new email and survey configurations based on language. Consignor has previously received complaints about emails using static text elements, resulting in non-English speaking customers receiving emails partially containing a language they don't speak. It was therefore important to not use any static text in neither emails nor surveys, allowing full customization of all text elements through the editors.

### **2.3.5 Security & Authentication**

Once integrated with Consignors system all authentication will be done through them. Because of this we chose to not spend a lot of time creating a secure solution, as for the final product our efforts would be largely wasted and we would like to spend our limited time elsewhere. We will instead be utilizing cookies for simple authentication purposes. Through a log in screen users will be able to provide a username, and by doing so will be given a session id which will be used for all calls to the API to identify the user. As mentioned this system is in no way secure, however, if Consignor wishes to expand on this authentication process it should be easy to do so as the entire system is built to support it.

### **2.3.6 Documentation and testing**

Due to Consignors expressed interest in further developing the software beyond the initial bachelor period, we decided early on that thorough documentation and broad test



coverage was of high priority. This creates a steady framework for anyone maintaining or continuing development of the project. To document our code we will be using the standard for C#, XML documentation [7]. This is created using simple and readable syntax, which the Visual Studio IDE uses to generate documentation with an xml structure. Below is an example of how one of our controller methods was documented.

---

```
1 <summary>
2 Receives a BrandingItem from the body of the request and passes
   ↪ it to the repository class, which attempts to add it to
   ↪ the database.
3 </summary>
4 <param name="item"> A BrandingItem object converted from Json.
5 </param>
6 <returns>
7 HttpUnauthorized if the given cookie is invalid.
8 HttpNotFound if the item is not found by the repository.
9 CreatedAtRoute if the item was added to the database.
10 </returns>
```

---

Documentation in this format alone is not very helpful, but with tools such as Sandcastle [8], recommended by Microsoft [7], this can be compiled into structured documentation. We will be working towards a 100% documentation coverage of the back-end, as well as a 100% test coverage of all controller methods.

### 2.3.7 Logging

In her book Dustin mentions that "One of the most common ways to increase testability of an application is to implement a logging mechanism" [9]. Logging provides detailed descriptions of the processing flow during execution, because of this logging is a great tool for testers and developers as it heavily reduces time spent on identifying problem areas.

Using Serilog [10] we will generate daily logs containing messages and exceptions produced by the runtime, as well as hard coded entries. These entries will be used to detail which function is currently being executed, the next step to be executed in addition to the state of any objects used within the function.

## 2.4 Product Backlog

The product backlog was hosted on Teamwork [11] which we integrated with our version control system on GitHub [12] such that all elements added to Teamwork also showed up in our GitHub issue tracker. Consignor was given access to the backlog on Teamwork, so that they always had full control over what was being developed each sprint. Teamwork allowed us to easily assign tasks to team members, providing time estimations, short descriptions of the task as well as the ability to record time spent on each sprint element. The final version of the product backlog can be viewed in appendix N.

## 3 Design

### 3.1 System Architecture

One of the goals of this project was to create a modern and scalable solution. Because of this we chose to go for completely separated frontend and backend solutions, communicating only via RESTful APIs. This would let a production environment scale different parts of the application differently based on the needs. Need faster API responses? Add more application servers. Need faster page loading? Add more web servers.

We initially made this sketch of the system architecture:

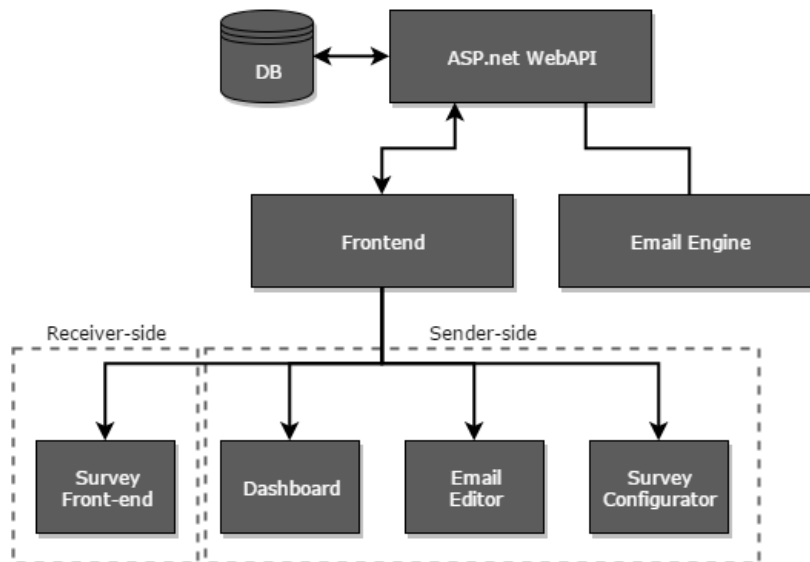


Figure 4: Diagram of system architecture.

We followed the design closely during development. For more information on the application stack see chapter 4 on page 27, or see chapter 6 on page 50 for more information on the server stack.

An important part of our design is the segmentation of frontend and backend, and by following industry standards like RESTful services we can ensure reliability and scalability. One of the bigger deviations in our application from a completely RESTful service is object naming. We have a service endpoint for each interface instead of for each object (except user).

After careful consideration and several iterations we came up with the following domain model to suit our needs best:

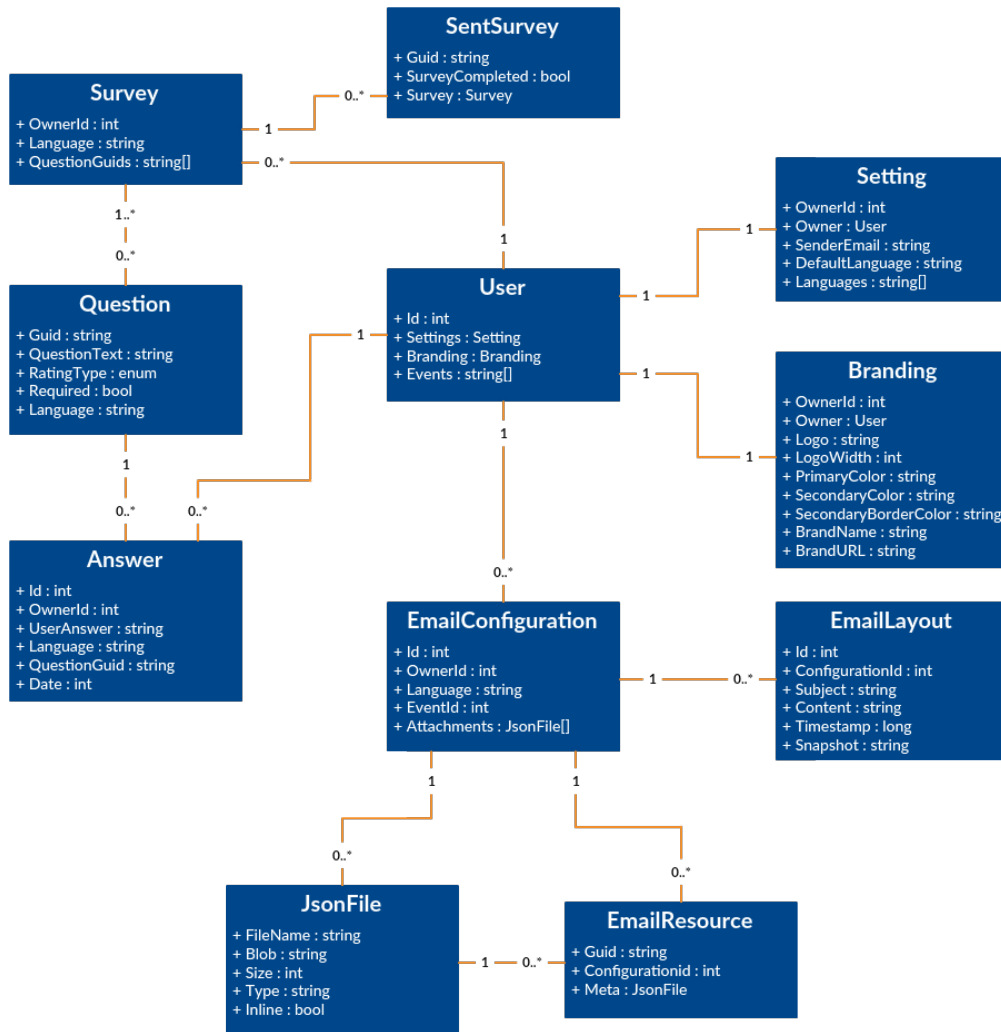


Figure 5: Domain model diagram.

Using this model has been very helpful during development to solve some of the more complex data interaction, particularly the functionality around Survey handling. Our implementation for the survey handling is based on the idea that history should never be lost, and thus question objects should never be deleted. Because of this, questions have a GUID based on the question text, and surveys will reference which questions it includes. Answers then reference a particular question, which always retains the original text, in addition to an owner. This means even if a user alters a survey, the questions and answers stay intact.

### 3.2 Sending Trustworthy Emails

The core of this application is creating and sending emails, however, sending emails that don't get caught by a spam filter is non trivial. A number of steps needs to be taken to ensure the most trustworthy email transmission possible, as described throughout this chapter.

We have considered two approaches for solving this problem:

**Scenario 1: Local MTA** Our server acts as MTA and is the origin of sent emails

**Scenario 2: Actor MTA** The actor sets up his own MTA, on his own server, and grants the application a login for the MTA.

This chapter is mainly describing scenario #1, as most of the issues described only matter for the first scenario. The application was designed to support both scenarios, more details on this in chapter [4.5.2](#).

### 3.2.1 Reverse DNS

The server sending the email needs to have a reverse DNS pointer set up to the domain sending the email. In our case we were planning on creating an application that could use multiple domains based on the actor, which doesn't really work with reverse DNS as it can only point to one domain. However, as long as the domain pointed to is trusted by the provider it is considered a safe origin for the email.

The reverse DNS is set by the ISP for the server, and in our case it was a matter of using the server hosts admin panel to set a pointer from the IP address of the server to 'mginx.tobbentm.com', our test-domain.

### 3.2.2 SPF

SPF, or Sender Policy Framework, is a DNS pointer defining which origins are allowed to send emails on behalf of the domain. As defined by RFC7208 [13], the SPF record is implemented using a specially formatted TXT record. This pointer can set an origin IP address directly, like we have on our test-domain:

---

```
mginx.tobbentm.com TXT "v=spf1 ip4:198.211.118.167 ~all"
```

---

Domains can also be set up to allow all the same origins as another domain, and as such add our test-domain to allow origins for sent emails:

---

```
actor.example.com TXT "v=spf1 include:mginx.tobbentm.com ~all"
```

---

This evaluates to adding all origins allowed by the included domains, to being allowed by 'actor.example.com'. If the actor is using a custom domain in our application, we need to ensure that the proper SPF records are in place. If the records are not in place, we may get caught in spam filters or potentially get flagged as a malicious origin and get placed in blacklists.

### 3.2.3 DKIM

DKIM, or DomainKeys Identified Mail, is a method of signing emails sent, to ensure no tampering has been performed on the email. Like with SPF, this is implemented by a specially formatted TXT entry [14] on a special subdomain called '{selector}.\_domainkey'. The selector can be anything that is a valid domain name, and is used when signing the

email, as a domain may have multiple keys for signing emails. The TXT entry needs to contain information about the key type, and the public key of the key pair. For example:

---

```
selector._domainkey.mginx.tobbentm.com TXT "v=DKIM1; k=rsa;
↪ p={public key}"
```

---

Then when sending the email we can choose which headers to sign in addition to the body of the email, and then add the signature as a separate header. We will be signing the following headers, in addition to the body:

- From
- To
- Subject
- Date

Using the example above, a DKIM signature in our case will look like this:

---

```
DKIM-Signature: v=1; a=rsa-sha256; d=mginx.tobbentm.com;
↪ s=selector;
c=relaxed/simple; t=1460461161; h=from:to:subject:date;
bh={signature}
```

---

In our testing environment we are using selector 'mginx'.

DKIM is not strictly necessary for passing spam filters, but it is definitely an added layer of proof, and many services like Google's Gmail, will acknowledge and verify DKIM signatures in order to further increase trustworthiness.

### 3.2.4 Encryption

Following standard industry practices [15] we will also encrypt all outgoing email from the application. We'll be using standard STARTTLS [16] encryption, using certificates from Let's Encrypt [17]. For more information about certificates and encryption, see chapter 6.2 on page 53.

## 3.3 Wireframes and frontend planning

At the start of the project we made a set of wireframes to represent the basic functionality and layout of the application, and came up with the following graphical style.

### 3.3.1 Wireframes

#### Landing Page

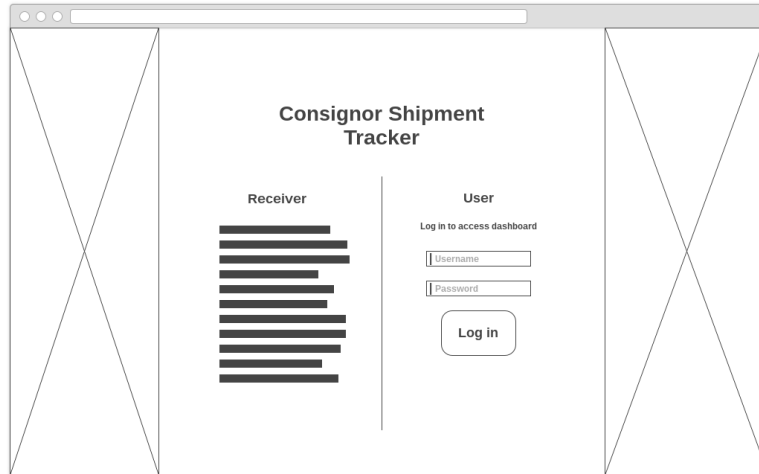


Figure 6: Wireframe of the landing page.

The landing page is just a simple page with information for receivers who manage to access the site directly, and a login for existing users. This was made before we came up with the name for the application.

#### Dashboard

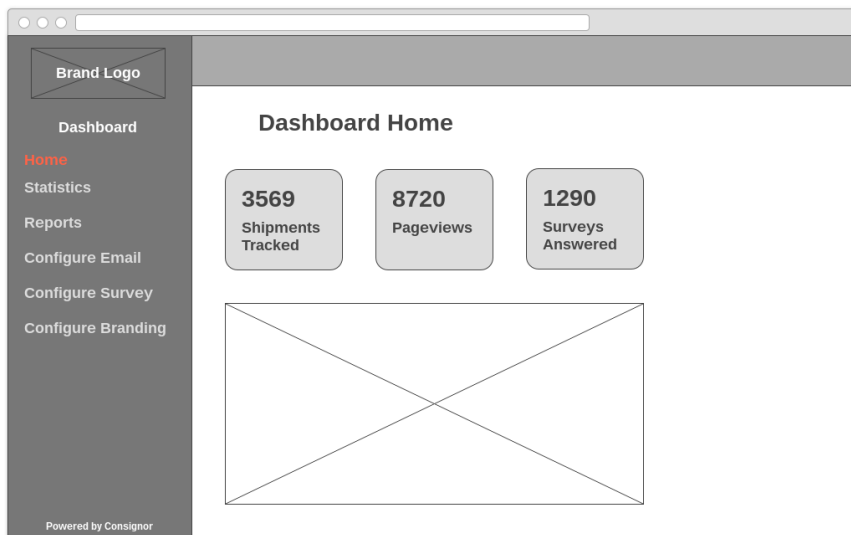


Figure 7: Wireframe of the dashboard.

The dashboard is a way to show brief statistics about the system and user responses, and a gateway to the rest of the functionality. This also defines the general layout of the

application, with a SPA-esque navigation pattern and main content in the right panel. The bottom main panel of the wireframe ended up being a notification tray to give a general system status. More details on this in chapter 4, section 4.4.2. The real estate for branding was originally thought to be the actors branding, but was cut to save space and to enforce a simple design for the application.

### Email Configurator (Editor)

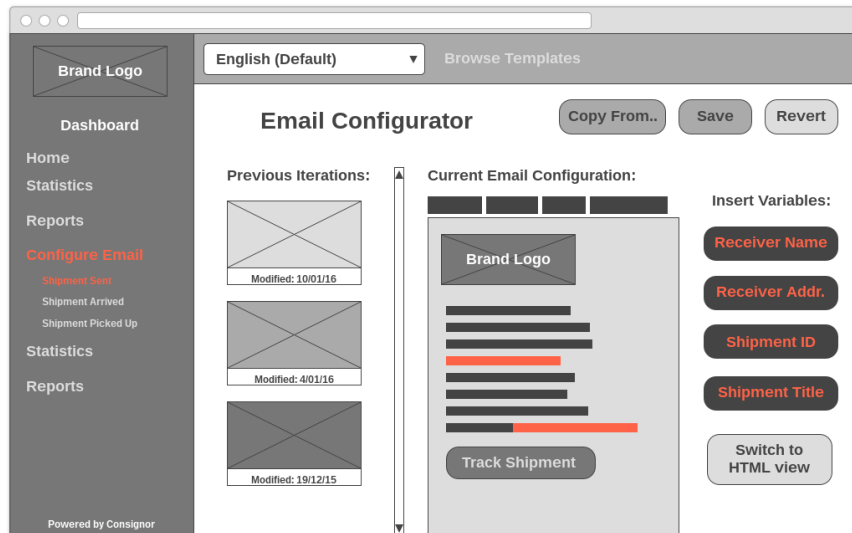


Figure 8: Wireframe of the email editor.

This is one of the first email editor concepts we came up with. It includes an easy way to change language, insert variables, edit the configuration and more. The general principles behind it made it through development, but was for the most part heavily re-designed. A lot of the controls were moved to toolbars or moved to a dialog toggled by buttons or actions. The iterations column was moved to the left hand side, so that the editor itself would be the centre of attention. Empty states were also added to give context when the actor did not have a configuration saved. We also added a lot of new functionality like previewing emails, more complex layout editing and more. For more details, see chapter 4, section 4.4.4.

## Survey Configurator (Editor)

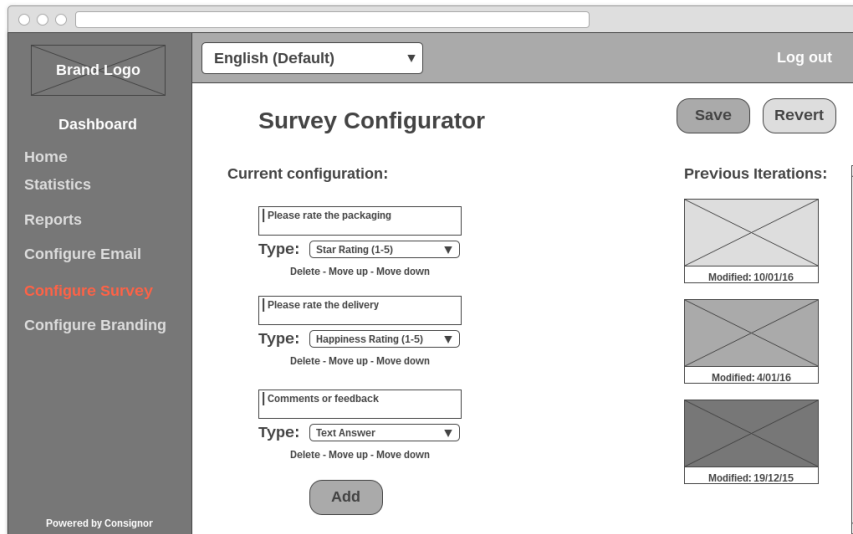


Figure 9: Wireframe of the survey editor.

As with the email editor, most of the principles depicted in the wireframe made it through development, with the notable exception of iterations.

## Survey Page

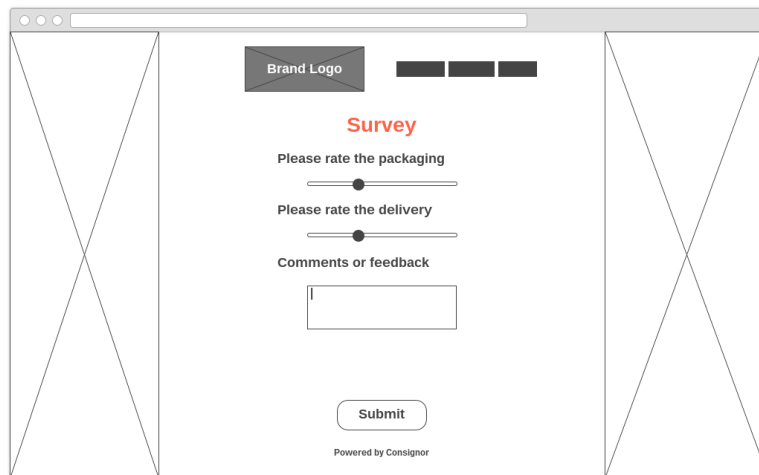


Figure 10: Wireframe of the survey page.

The survey page is a relatively simple page, with injected branding from the actor that owns the survey, and questions loaded from the most relevant language.

### 3.3.2 Graphical style

During the first planning meetings with Consignor we were encouraged to follow the existing styles and patterns of Consignor products. This is part of the reason we planned



on using Bootstrap for layout, with the font *Maven Pro* and main color *#166dbe*.



Figure 11: Font and color representation.

During the design phase we got a lot of inspiration from the Consignor Portal product, in regards to layout, navigation and more.

### 3.4 File structure

#### 3.4.1 Application Organization

The application itself is organized based on feature, with a couple of root folders for general features, and a folder for the API with each endpoint in its own folder within. The endpoint folders have 2 folders each, one for the controllers, and one for all DAL classes, as illustrated in the following diagram:

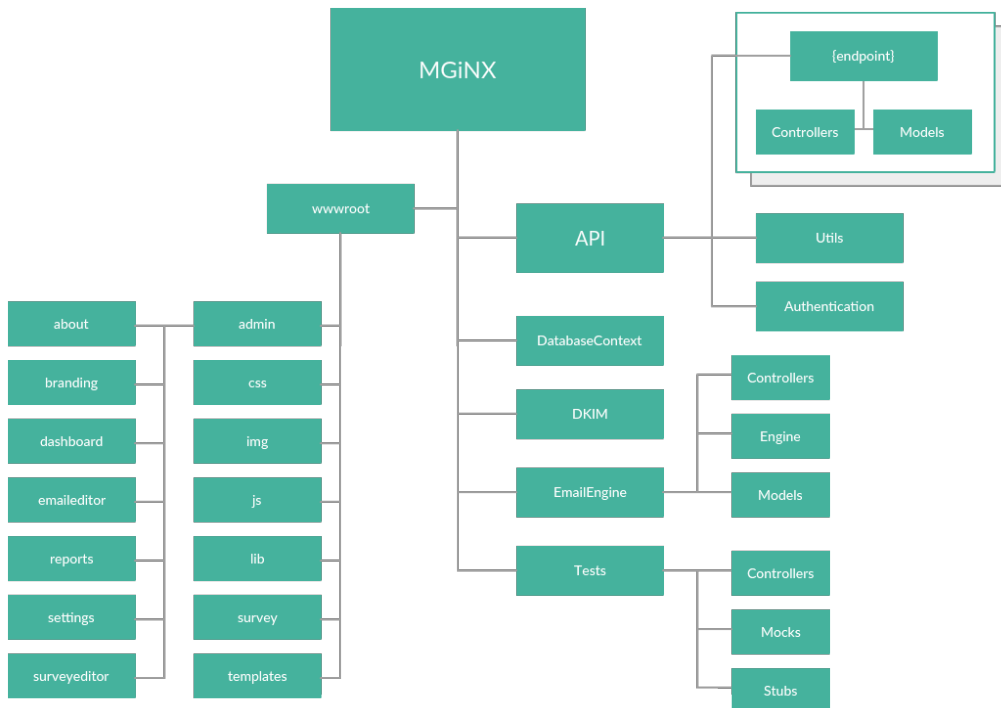


Figure 12: Diagram showing file structure of application.

Based on wireframes and requirements, we came up with the following 7 endpoints needed for the API:

- Branding
- Dashboard
- EmailLayout
- Reports
- Setting
- Survey
- User

There are also some endpoints exposed for testing purposes like creating new shipments. These are exposed in separate controllers in root folders.

### 3.4.2 User Data

While most of the user data is stored in the database, there are some files stored on disk directly. The root of these files are located in `/app/userdata`. For more details on this see chapter [6.1.4](#).

#### Attachments

Each user will have it's own folder under the root attachment folder which contains folders for all events. Each of these folders contains the languages used for the specific event and user. The language folder is what contains the actual attachments for the given language, event and user.

An example of this would look like this:

---

```
/app/userdata/attachments/{user-id}/{event-id}/{language}/file.ext
```

---

#### Static Resources

Separate from attachments are static resources. This is different from attachments in that they are linked to the email without being attached. These resources are quasi-independent from users, events and languages, and are mapped using GUIDs. The path to these files will be something like this:

---

```
/app/userdata/resources/{GUID}/file.ext
```

---

## 4 Implementation

### 4.1 Tools & Codebase

For developing the application we used the Visual Studio 2015 IDE, the recommended and most common IDE for .Net projects. This version of Visual Studio support the new .Net runtime, as well as with packet managers, making project management and build automation easier.

For backend packet management we used NuGet [18], which ties nicely in to Visual Studio, and also our server build process. Our complete configuration can be seen in appendix J. For frontend packet management we used Bower [19]. Our complete configuration can be seen in appendix K.

For working together as a team, we used Git [20] as our version control system, with GitHub [12] as the repository host. As discussed in chapter 2.4, the GitHub repository got integrated with the Teamwork project to automatically generate issues based on the backlog.

In terms of size, the codebase ended up being relatively compact:

<b>Backend</b>	68 C# Classes 4703 Lines of code 1027 Lines of comments 64 Unit tests
<b>Frontend</b>	3313 Lines of JavaScript code 2529 Lines of HTML markup 784 Lines of CSS markup

Which goes to show the power of the framework we are using, as discussed in the next chapter.

### 4.2 ASP.Net Core

While planning the project we communicated closely with Consignor to determine our software stack. Consignor is largely a .Net based company, which resonated with us as we wanted to learn .Net as well. With the new version of ASP.Net just being launched, we decided to use ASP.Net Core RC1 for this project. Compared to the 'old' ASP.Net 4.6, Core is faster and leaner, with support for self hosting on Linux and in Docker containers. We did hit a few bumps in the road however, as is often the case with early version release candidates, discussed more in chapter 4.2.2.

The way ASP.Net is structured is illustrated in figure 13, and it is the way we are doing it as well, with a clear separation of responsibility between controller, repository and

databasecontext. The structure is based on principles from other software patterns like MVC, where separation of concerns plays a major role in improving code readability and maintainability.

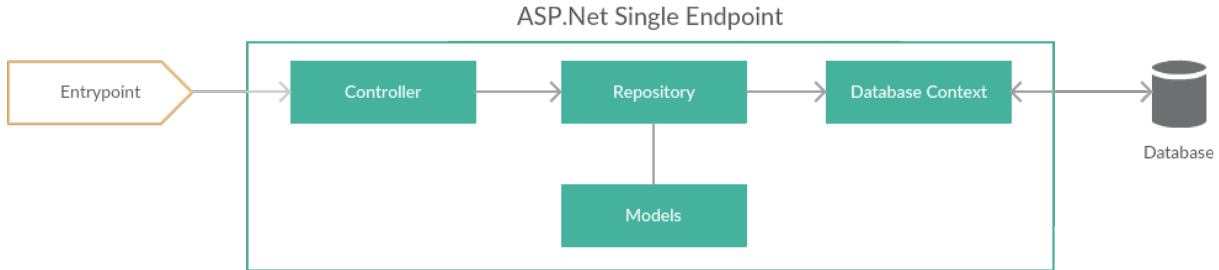


Figure 13: .Net Architecture Diagram.

An example of our typical controller can be seen in appendix L, and the repository it uses in appendix M.

#### 4.2.1 Entity 7 ORM

With a new version of ASP.Net comes a new version of Entity Framework as a ORM. As with ASP.Net, this is also in a pre-release stage of development. One of the bigger limitations posed on us is the availability of database connectors. When we started the project, only two database connectors was officially supported: Microsoft SQL Server and SQLite. Since we were running our test environment on a Linux host, Microsoft SQL Server was not relevant. And SQLite by nature is not optimal for our scenario. Luckily, a third party connector was also in progress, called Npgsql [21]. This connector allows connections to PostgreSQL [22] databases, which is the best solution at the moment. PostgreSQL is a highly performant and standards compliant SQL database. It is also cross platform, running on our chosen operating system, Linux.

The ORM itself supports both code-first and database-first approaches, and we chose code-first. This means we made all the models for the application, and then created a custom context class describing the models with keys, relations and values. Then using Entity Framework, we compiled migrations and ran them on our database to create the schema.

#### 4.2.2 Release Candidate troubles

When we started the project, RC1 was the newest version of ASP.Net, with RC2 originally slated for release in mid-march. Most of the issues described here has been solved in RC2, but unfortunately we are still waiting for the RC2 release with the completion of this thesis.

#### Linux Socket Bugs

When using the Core Runtime (CLR), the native socket implementation would sometimes time out [23], causing particular problems for our Postgres database connector [21]. The solution to this was manually adding some linux-specific libraries

---

```

1 {
2   ...
3   "dependencies": {
4     ...
5     "runtime.linux.System.Net.NetworkInformation":
6       ↪ "4.1.0-beta-23516",
7     "runtime.unix.System.Net.Security": "4.0.0-beta-23516"
8   }

```

---

As well as replacing the compiled system library 'System.Native.so' when building our Docker container (see chapter 6.1.4). The newly compiled version can be found in the Github issue [23], and has reportedly been fixed in RC2 of .Net Core (still in progress/un-released).

### Kestrel HTTPS Issues

Since we wanted to encrypt the application endpoints we needed HTTPS support in the web-server. Unfortunately, the built in Kestrel web server in ASP.Net does not support HTTPS encryption. However, we can get around this by setting up a proxy to encrypt the backend, and that's what we did. Using an nginx web server as a proxy with certificates (see chapter 6.1.2) we successfully encrypted all static content. The API however, was not working correctly. We would get the content from the controller, but the request is never closed, and thus times out in our javascript code. This is also reportedly fixed with RC2, and still a problem for us. As a temporary workaround, we are overriding the proxy. This issue has been reported on Github [24].

## 4.3 Database

Since we were able to start this project on a clean slate, we decided to build our database using a code-first approach. This means we define models and relations using POCOs (Plain Old C# Object) for models and defining relations using Entity Frameworks Fluent API [25]. Using the Entity Frameworks tooling, we can then compile a migration to run on the database to create the schema based on the models and relations we have defined. The following is an example of one model, in this case for the user:

---

```

namespace MGiNX.API.User.Models
{
  public class UserItem
  {
    public int Id { get; set; }
    public SettingItem Settings { get; set; }
    public BrandingItem Branding { get; set; }
  }
}

```

---

This item needs three things defined: Primary key, relation for SettingItem and relation for BrandingItem. We could do this by using C# attributes, or by using the Fluent API. We settled on using the Fluent API for the project, in order to have all definitions in one

place. The definitions for this item would look like this:

---

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<UserItem>().HasKey(m => m.Id);

    builder.Entity<UserItem>()
        .HasOne(u => u.Branding)
        .WithOne(b => b.Owner)
        .HasForeignKey<BrandingItem>(b => b.OwnerId);

    builder.Entity<UserItem>()
        .HasOne(u => u.Settings)
        .WithOne(s => s.Owner)
        .HasForeignKey<SettingItem>(s => s.OwnerId);

    base.OnModelCreating(builder);
}
```

---

This example sets the `Id` variable to be the primary key, as well as defining one-to-one relations between this item (`UserItem`) and the two related items (`BrandingItem` and `SettingItem`). The snippet is taken from the database context, and is used when compiling the migration. The Fluent API also lets us add other properties to objects, and we are using this to inject shadow properties like timestamps into objects to keep track of changes done to them:

---

```
builder.Entity<UserItem>().Property<DateTime>("UpdatedTimestamp");
```

---

We can then override the base `SaveChanges()` to update the property upon saving.

## 4.4 Frontend

### 4.4.1 Intro

The frontend is primarily built with Bootstrap [26] and jQuery [27]. As discussed in chapter 3.3.2, we are using fonts and color schemes inspired from Consignor Portal [28]. A number of libraries make up minor functionality for the frontend, for a complete list see appendix K on page 107.

#### 4.4.2 Dashboard

The dashboard gives the user a brief overview of the system status, prioritizing any error messages if the system is not configured correctly. As seen in figure 14, the layout is rather clean and only two pieces of content.

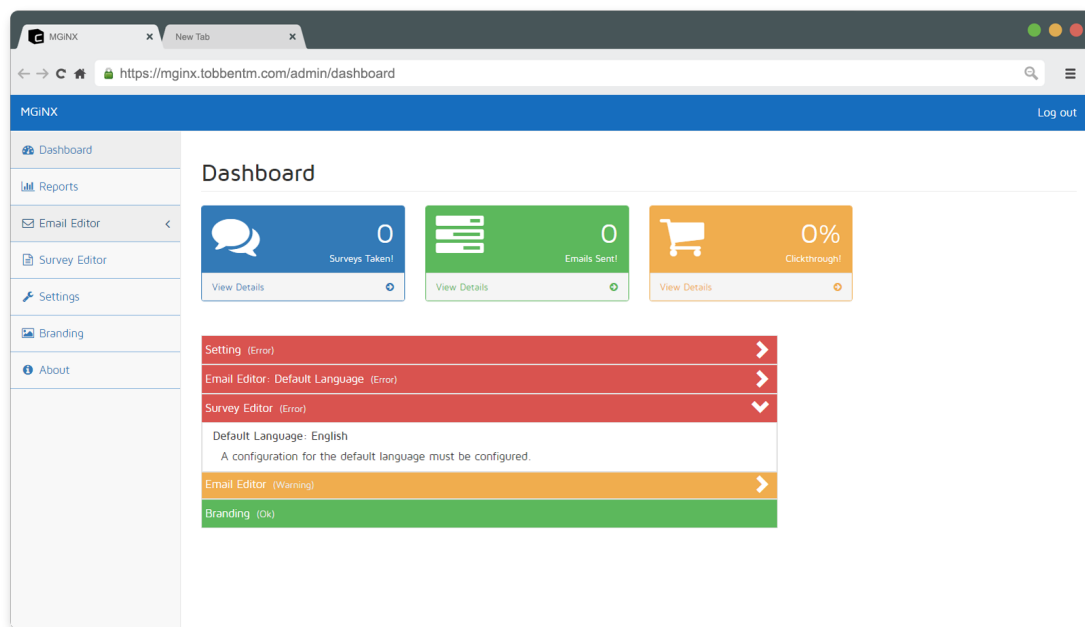


Figure 14: Screenshot of application dashboard.

The top three boxes have the following purposes:

**Surveys Taken** Shows the number of surveys submitted by receivers

**Emails Sent** Shows the total number of emails sent for the actor by the system

**Clickthrough** Shows the amount of emails where the receiver has interacted with the email (either clicked tracking button or survey button)

Unfortunately, we did not have time in this project to fully implement the backend statistics for this.

The lower part of the layout lists system messages with the general system status. Red messages are error, and effectively hindering the system from sending emails. Examples of this is no configuration for default languages, or no branding set up. Yellow messages are warnings, and ultimately not hindering the system from sending emails. This can be missing configuration for secondary languages, in which case the system falls back to the default language. Green messages indicates everything is OK with a given module.

### 4.4.3 Reports

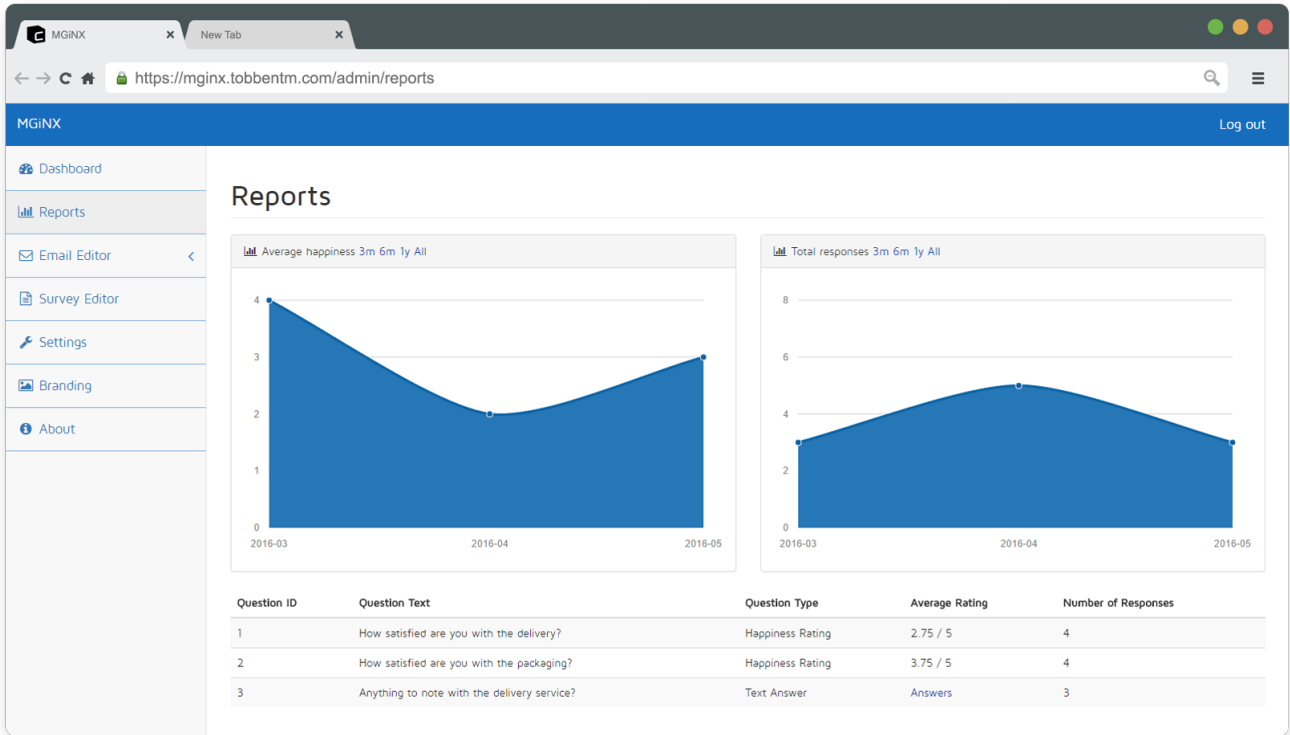


Figure 15: Screenshot of report.

On the top portion of the screen there are two graphs. These display monthly average rating received from surveys, and total amount of surveys answered. They can also be individually adjusted to show the latest 3 or 6 months, as well as the last year and total. When the user first loads the report page the full data set for both graphs are fetched, and parsed by the browser when the user wants to change the range displayed.

The bottom part shows a list of all survey questions. This includes all questions that are a part of the users current survey configuration, in addition to questions that have been in previous iterations. This decision was made as historical data can still be valuable, however, to keep it organized all questions are sorted from newest to oldest. There are two types of questions, rated questions and text questions. Rated questions are simple to load as only the average is displayed, which can be calculated on the server. However, with many text based questions, loading all of these answers at the same time is not going to be a sustainable solution. To solve this, text answers will only be fetched on a per request basis. For each text based question a link will be available which will fetch all answers for that particular question and displayed in a separate window.

#### 4.4.4 Email Editor

The email editor is part of the core functionality of the application. The following screenshot illustrates what you would see the first time entering the application, and the editor.



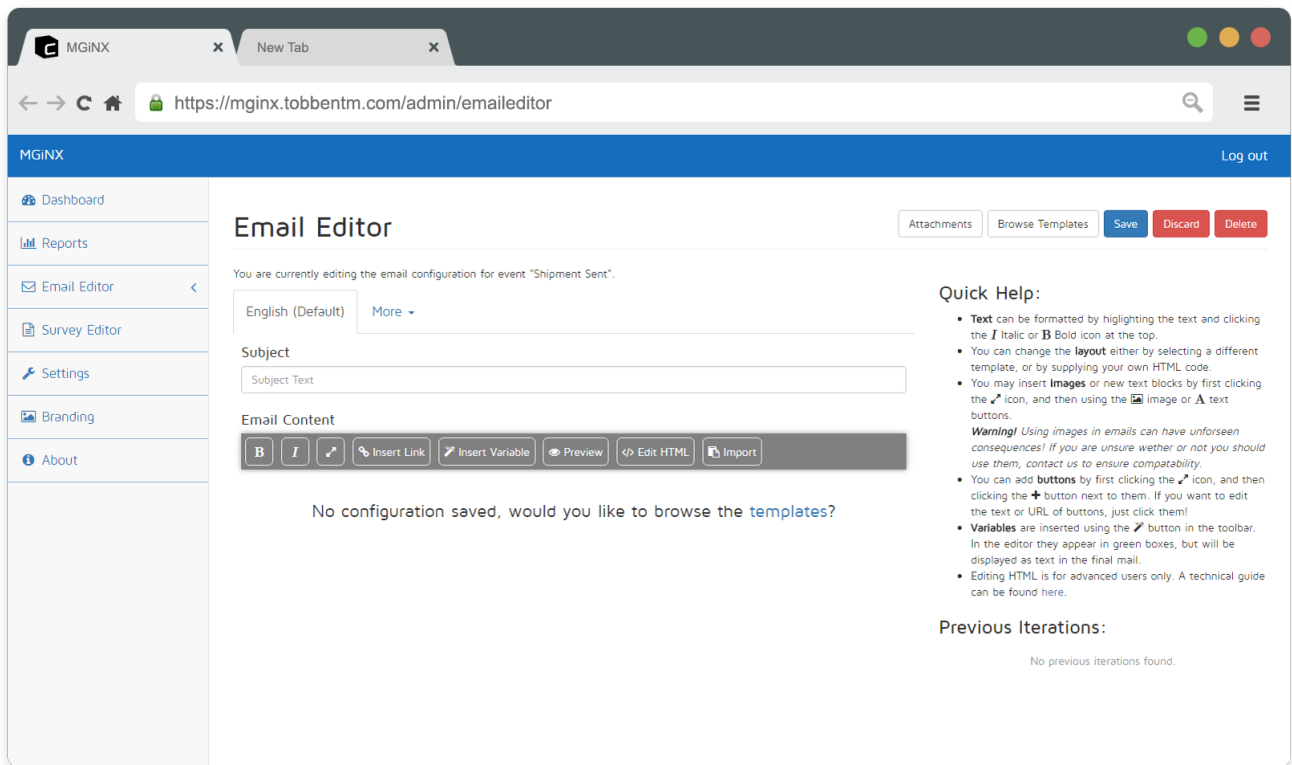


Figure 16: Screenshot of email editor.

Since no configurations have been made, the editor is in an empty-state, hinting the user to browse templates to get started.

In line with the requirements, the user can choose which language to edit, as well as adding new languages under the 'More' tab. The language area will also automatically open the default language as well as show which it is.

In the right column of the editor we have a row of buttons:

**Attachments** Opens a dialog where the user can upload static attachments to be included in every email sent in this language with this event. The dialog also allows for previewing and deleting attachments. The files are stored according to the format explained in chapter 3.4.2.

**Browse Templates** Opens a dialog displaying available templates the user can download to quickly get started.

**Save** Saves the configuration.

**Discard** Reverts any changes made since the last saved configuration.

**Delete** 'Deletes' the current configuration. No configurations are actually hard-deleted, but the configuration will not have effect on email compilation, and the editor will show no configuration active.

Under the buttons we have added a quick help section, highlighting some of the functionality of the editor, in case the user forgets. There are also some warnings for less experienced users, to promote lean and compatible email layouts.

Underneath the quick help is a list of previous iterations. By default, this show up to the five last configurations made, to let the user revert to any iterations he wants.

### Editor Core

The editor is mainly using the HTML Content Editable [29] feature to allow the user to edit the HTML contents directly. To help us with this task we are using a library called Medium.js [30]. This library wraps the contenteditable methods in a better editor object, and has some methods for wrapping content in tags amongst others. Much of the editor functionality is delivered through a toolbar:

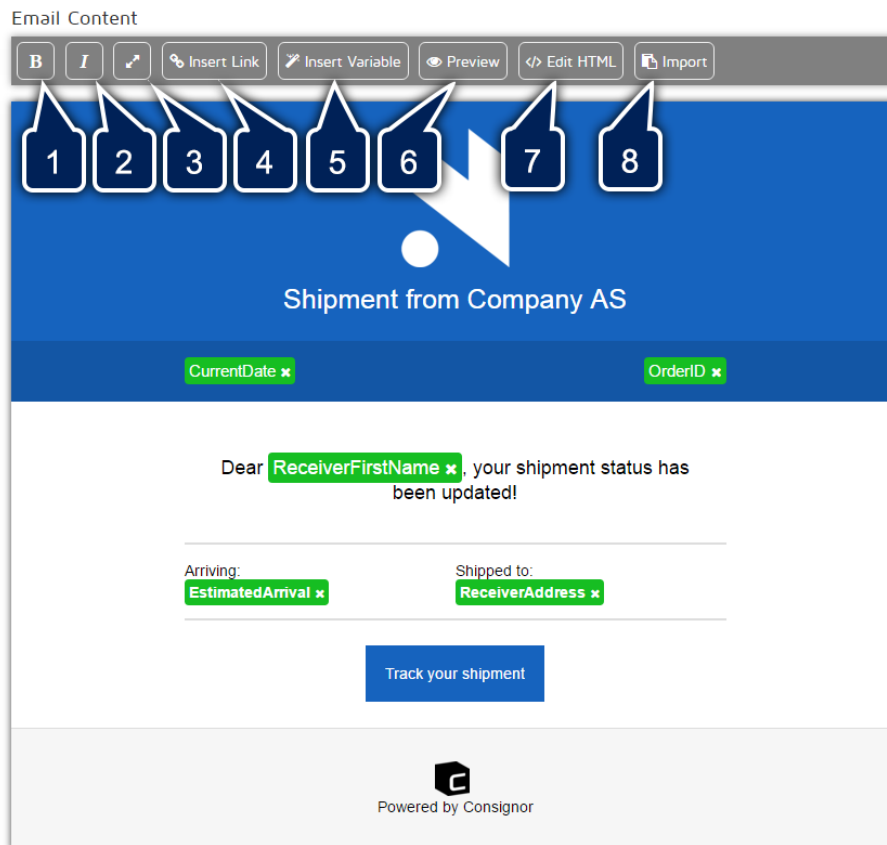








Figure 17: Screenshot of email editor with toolbar and content.

With this toolbar the user has access to almost all of the tools the editor has to offer. The buttons have the following functionality:

1. **B**: Button to wrap the currently selected text in '<strong>' tags, making the text **bold**.
2. *I*: Button to wrap the currently selected text in '<em>' tags, making the text *italic*.

3. : Button to show expanders in the editor, explained in detail in chapter 4.4.4 on page 35.
4.  **Insert Link**: Opens a dialog with two fields; *link text* and *link URL*. If user has selected text, the *link text* field is automatically filled out with the selected text, and when inserting the link, it will replace the selected text, in effect making portions of the text a hyperlink. If the user has not selected any text before clicking the button, the link will be inserted wherever the users cursor was when clicking the button.
5.  **Insert Variable**: Opens a dialog containing a selection of variables. Works like the link insertion in that the variable will be inserted wherever the user had his cursor when clicking the button. The variables themselves are explained further in chapter 4.4.4 on page 36.
6.  **Preview**: Opens a new browser window and displays the email layout with the variables replaced with static text. Useful for seeing how it will really look, without any editor markup.
7.  **Edit HTML**: Replaced the GUI-based editor with a text field containing the HTML code for the layout. Any changes made in the HTML is reflected when changing back to a GUI-based editing mode.
8.  **Import**: Opens a dialog where the user can select to import an existing layout from another event and/or language. Very useful when translating.

## Expanders

In order to provide more customizability to the user other than formatting text, we implemented the functionality to alter the layout by adding new paragraphs, images or buttons. As described in chapter 2.3.3, there is a need for the markup to be compatible with most mail readers on the market. Because of this, we needed to limit what the user was able to do, and as such we only allow adding new paragraphs in certain places, based on the template. When enabling the expanders template #1 looks like this:

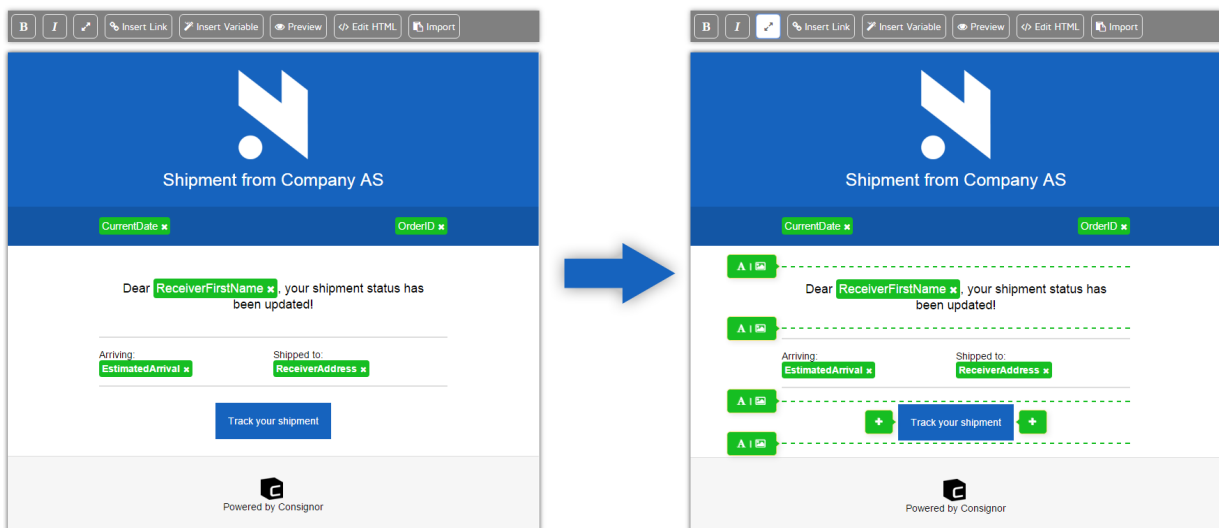





Figure 18: Screenshots of email editor with and without expanders.

**Buttons** New buttons can only be added next to existing ones. This is to preserve styling and appearance. When clicking the  popup next to existing buttons, a dialog will appear, where the user can define the text and URL of the new button. The same dialog appears when click existing buttons, allowing for intuitive editing of existing buttons. The button dialog also gives the option for special URL, and we have defined two: *#tracking* and *#survey*. These will be changed when compiling the email to either shipment tracking, or to a generated unique survey for the shipment.

**A Text paragraphs** By clicking the **A** button, the editor will copy a row and insert wherever the separator is located. Like with the buttons, this will preserve styling and appearance in order for it to look natural. The new paragraph is inserted immediately, without a dialog asking for content, instead using the native editor workflow of editing directly.

 **Image** Very similar to the text paragraph functionality, by clicking the  button the editor will ask the user via dialog for image details, and then insert the image into a row with the same styling as neighbouring rows. The dialog lets the user either define a location for the image, or upload to our application. By uploading the file gets stored with a GUID as described in chapter 3.4.2. The user may also define a URL for the image, effectively wrapping the image in a anchor tag and linking to the URL.

## Variables


As each email needs to have shipment-specific information in them, or just dynamic data, the editor allows for special tokens to be inserted into the email configuration. These are then replaced by the email engine with data from the shipment data payload. In the editor the tokens act as single entities and can therefore not be changed, other than being deleted completely. This is done by have the tokens not be content editable:

---

```
<td class="content" contenteditable="true">
  ... <span class="data-variable"
    ↔ contenteditable="false">TokenName</span> ...
</td>
```

---

The tokens also get injected with a button to delete itself, though it can also be done using the editor and deleting it using backspace/delete to delete it like if it were a character.

The tokens are organized in a dialog that gets shown when clicking the  **Insert Variable** button. The dialog is shown in figure 19, with the currently implemented tokens. As mentioned in chapter 2.2, we are using a reduced and modified json payload to show proof of concept, so not all json values have been listed as tokens.

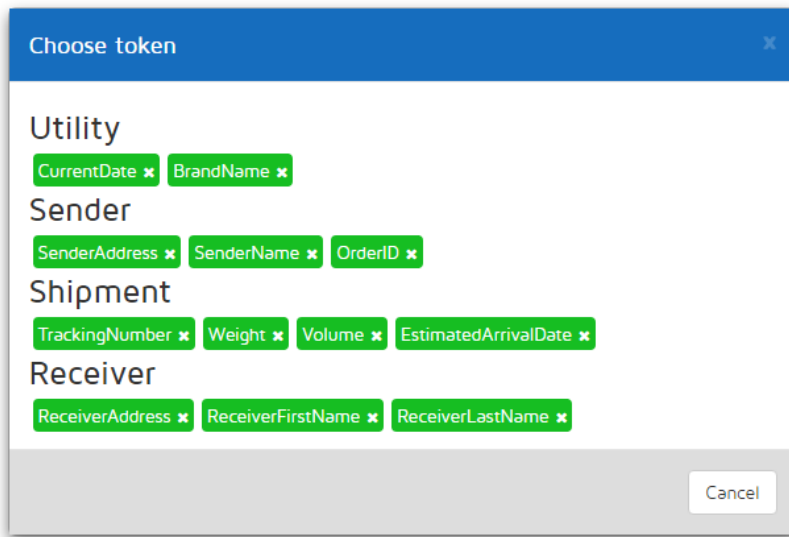


Figure 19: Screenshot of variable dialog.

In figure 19 we have sorted and categorized the available tokens. The Utility tokens are system generated at the time of sending/compilation, while the rest are taken from the json payload.

#### 4.4.5 Survey Editor

The actors also need a way to easily define the surveys to be sent out, and for this we have the survey editor:

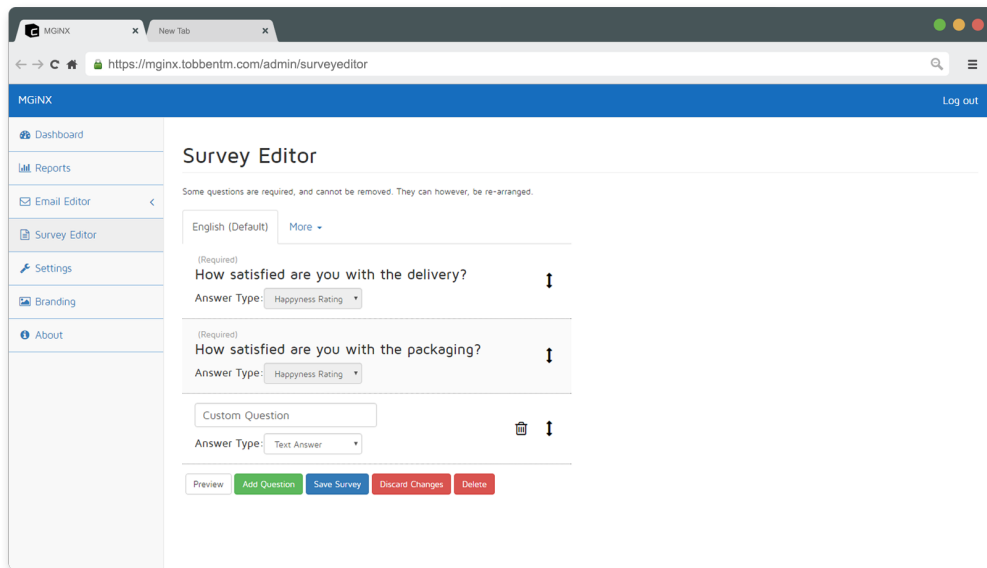


Figure 20: Screenshot of the survey editor.

These are standard questions to assure that the user gets at least a minimum level of feedback. Initially, the actor will only see the system required questions, but he can easily

add his own, rearrange questions or delete old custom questions. Questions can have one of three different formats:

**Happiness Rating** Essentially a rating of 1-5 smiley faces

**Star Rating** Same as happiness rating, only with stars

**Text Answer** Recipient can write a custom response

As a handy way to see what the survey looks like before committing, the actor can easily preview the current survey configuration by clicking preview. This will open a new browser window with a survey template injected, along with the actors branding. All other buttons should be self explanatory.

#### 4.4.6 Branding

As a way to unify all outgoing media from the application, we have a page where actors can define the branding they want to display for all media. This includes configuring brand naming, color schemes and logo, as displayed in figure 21.

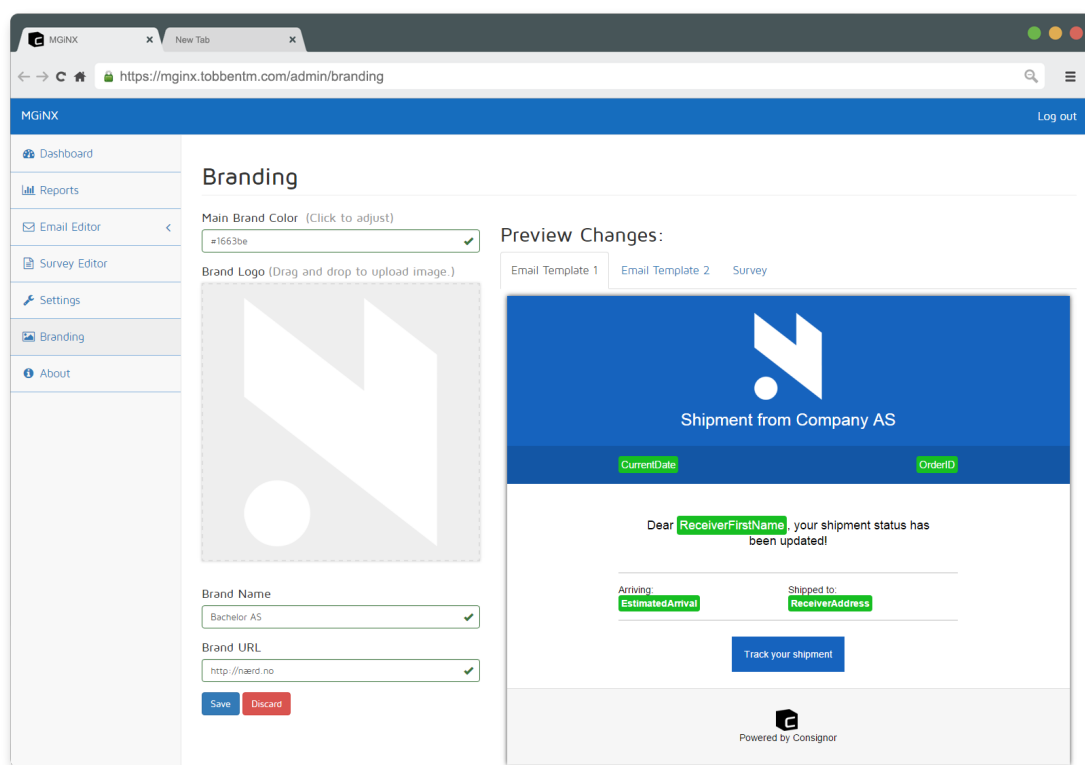


Figure 21: Screenshot of branding options.

The color scheme, logo and brand URL gets injected in all emails, as well as survey pages. The brand name is user when sending emails as the email sender. All changes made on this page are reflected live in the preview column to the right, making it easy to tune the branding and getting immediate feedback.

#### 4.4.7 Settings

A fairly simple settings page, in order to further customize the application to the current actor. We had planned on exposing two major settings here: language and email setup. From this page the actor can add languages, change default language for all emails and survey configurations as well as delete entire languages (with every configuration affiliated). Unfortunately we did not have time to fully implement custom email settings, more on this in chapter 4.5.2 on page 43.

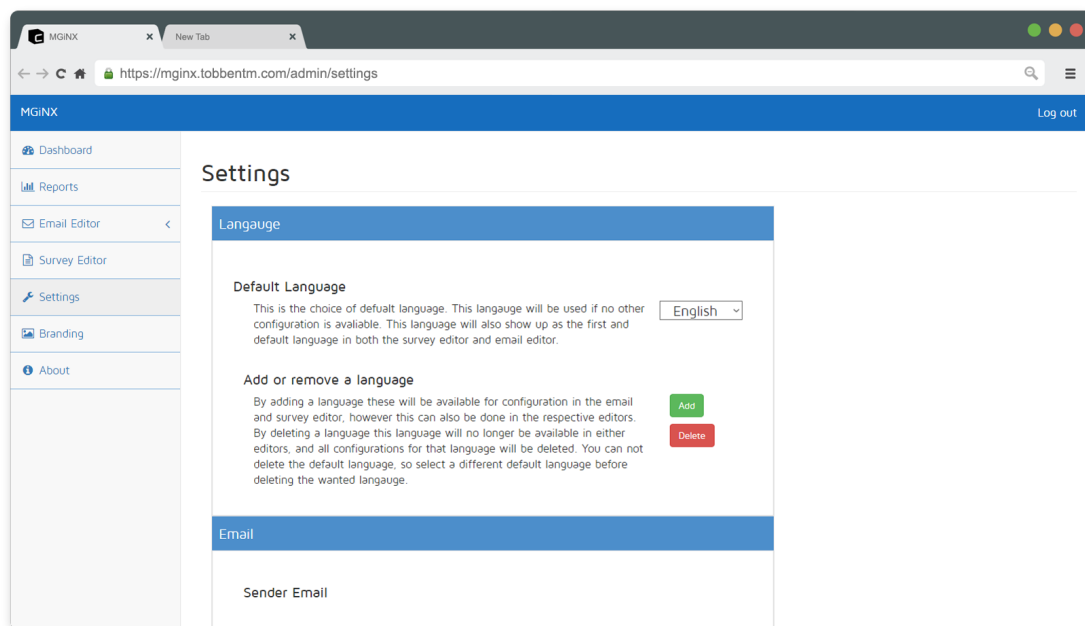


Figure 22: Screenshot of application settings.

#### 4.4.8 Survey

As the only receiver-facing page, it's important the the survey page looks nice, scales nicely to different devices and represents the actor appropriately.

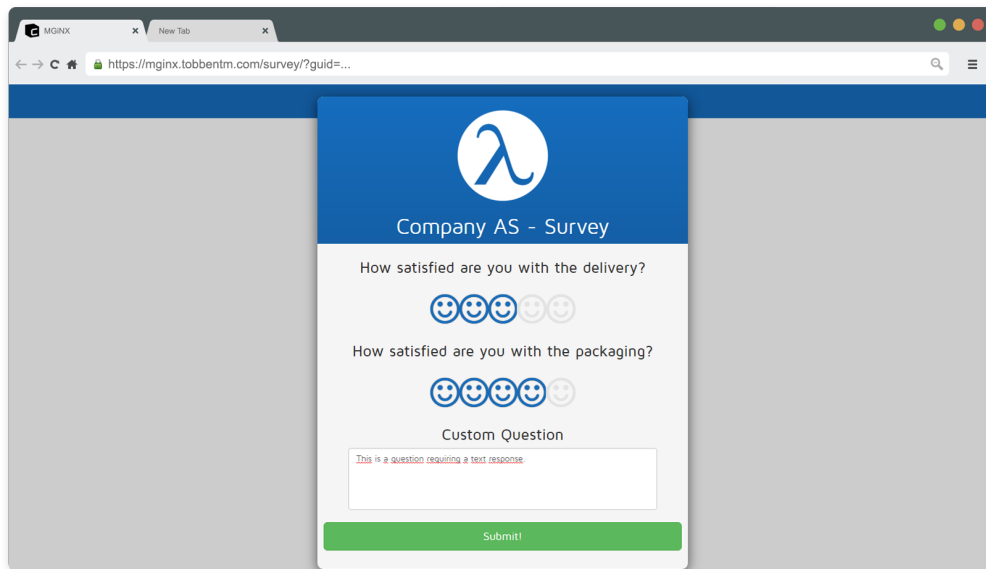


Figure 23: Screenshot of survey page.

This page will request a survey from the server, based on the URL, and inject the quasi-dialog with questions, as well as theming the page. If the actor has defined a different color scheme, changed logo or name, it will be reflected in this page. The questions will be taken from the actors current survey configuration, for the particular language the shipment is based on.

To ensure we don't get inflated survey results and only package receivers have access to this page, we will generate one time access keys that are appended to the survey URL sent to receivers. These keys are stored in the database and are used to authenticate that the person attempting to access the website is indeed a registered receiver. If the key is present and valid the correct survey will be displayed, if not the user will be denied access. Once a key has been used to answer a survey that key is no longer valid and will be removed from the database.

## 4.5 Backend

### 4.5.1 API Endpoints

In order to serve the frontend with data, we need a whole set of endpoints for the interface between the two components. As mentioned in chapter 3.1, we have primarily created endpoints for each frontend page, along with some general/utility endpoints. The following list contains a brief description of each endpoint:

**Branding** (`../api/branding`) Responsible for updating the branding profile for a given actor. Also handles image upload for storing actor logo.

**Dashboard** (`../api/dashboard`) Responsible for creating system status messages for use in the dashboard.

**EmailLayout** (`../api/emailayout`) Responsible for all things related to the email editor,



including:

- Email configuration retrieval, updates and deletion
- Attachment uploading, previewing and deleting
- Template listings

**Report** (`../api/report`) Responsible for generating reports and statistical data for the reports page.

**Setting** (`../api/setting`) Responsible for updating the setting profile for a given actor.

**Survey** (`../api/survey`) Responsible for both the survey editor functionality, and individual surveys. For the survey editor, it delivers the ability to retrieve, create, update and delete configurations, and with individual surveys it allow the browser to get survey questions and submit answers.

**User** (`../api/user`) Responsible for handling user (actor) profiles. General endpoint, used by many of the pages.

**EmailEngineAPI** (`../shipment/new`) Endpoint for generating email, hooks into the email engine, described in the next chapter, [4.5.2](#).

The exhaustive list of endpoints is listed in appendix [O](#), as discussed in chapter [5.4](#).

#### 4.5.2 Email Engine

The email engine of the application is responsible for inflating an email, and sending it. It is called whenever a new shipment is received by a controller. The engine goes through a number of steps to produce and send the email:

1. Extract meta information from json payload. (Installation ID, Event ID, Destination Country)
2. Retrieve database objects based on meta info. (User object, email configuration and layout)
3. Optionally fall back to default language if destination country is not default, and not configured
4. Compile the email layout:
  1. Parse layout to get a representation of the nodes
  2. Replace all Utility Variables with system generated values
  3. Replace all other Variables with payload data (or fallback to "??")
  4. Inject branding (Color scheme, logo, URL)
  5. Inject link for tracking shipment
  6. (Optional) Inject link for survey

This returns a compiled layout with images inlined and everything ready.

5. MUA Compiles the email Mime message:
  1. Add all headers based on actor data (sender email address, sender name, subject) and payload information (recipient email address)
  2. Extract all images inlined in layout and move to inline resource (linked locally in HTML)

3. Add compiled email layout as body
4. Add all static attachments as Mime attachments
5. Encode Mime message using Qouted Printable encoding
6. Sign Mime message and headers using DKIM key and add signature as header
7. Connect to email relay and transmit Mime Message

Based on this general overview of steps taken, the finished email should consist of many Mime parts making up the body, inline resources and regular attachments. For testing purposes the controller invoking the email engine returns the compiled email, as well as some metadata about the email. For help making the Mime message we are using a library called MimeKit [31]. This way we can incrementally build a Mime message by adding HTML content, text content, inline resources and attachments. We can then encode the message using Qouted Printable encoding. This ensures compliance as defined by RFC2822 [32], and prevents the email from being forcefully modified by other agents for compliance reasons. One of the main reasons for encoding is to ensure a line length of maximum 78 characters. The library also lets us sign the message using a DKIM key-selector pair. The following demonstrates the signing:

---

```
var headers = new [] { HeaderId.From, HeaderId.To,
    ↪ HeaderId.Subject, HeaderId.Date };
var signer = new DkimSigner("/app/DKIM/mginx.private",
    ↪ "mginx.tobbentm.com", "mginx");

// Sign body with DKIM signer
message.Sign(
    signer,
    headers,
    DkimCanonicalizationAlgorithm.Relaxed,
    DkimCanonicalizationAlgorithm.Simple);
```

---

As seen in the example, we are using the domain "mginx.tobbentm.com" and selector "mginx".

For actually transmitting the Mime message, we are using MimeKits sister library, MailKit [33]. Using this we can connect to a given host, optionally authenticate and transmit the email.

---

```
using (var client = new SmtplibClient())
{
    client.Connect("dockerhost", 25, SecureSocketOptions.None);
    client.AuthenticationMechanisms.Remove("XOAUTH2");
    client.Send(message);
    client.Disconnect(true);
}
```

---

This example illustrates connecting to the local host MTA, where we do not need to authenticate in order to relay email. Alternatively we could connect to an actors custom MTA, and authenticate using actor settings instead.

## Custom Email

From the beginning we had planned on allowing the actor to define a custom email to send emails from. During our research (see chapter [3.2](#)) it was clear to us that this needed two things from the actor: SPF and (optionally) DKIM set up for us. And we initially developed the system to accommodate this, however we hit a challenge we did not manage to overcome; a lack of DNS functionality. For an actor to be able to use a custom email address, we felt the need to actually verify the setup, and use a local email address until it is set up. With the ASP.Net Core project not being fully grown up, there was no system library or third party libraries which delivered fully functional DNS lookups. We considered writing our own DNS client implementation, but ended up rather prioritizing other functionality. Later in the project we also discussed the option of letting the actor set an email relay as an option to this, though we faced the same issue of time prioritization. The email MUA in our system is fully capable and ready for custom email relay or DKIM settings though, so once we have the ability to verify DNS pointers it would not be much more work.

## 5 Testing and Quality Assurance

### 5.1 Unit Testing

In the early stages of our planning phase we recognized the importance of having full test coverage of the API controllers as this functions as the backbone of the entire system, and hence chose to dedicate an entire two week sprint to fully test the back-end. Over the course of this two week period we wrote unit tests that covered all current controllers, as well their fail states.

Xunit [34] was used to write the unit tests, however, we also needed a mocking framework due to all controllers heavily interacting with the repository classes. The repository classes were made to divide database operations from the rest of the system and primarily consists of Entity Framework 7 code. We did not want to spend the time or resources needed to write unit test for these repositories as we viewed this with very little importance, so instead we chose to mock them. Working with a release candidate version of ASP.Net Core made it very difficult to find a mocking framework that had the functionality we needed, as very few currently support .Net Core. In the end LightMock.vNext [35] was chosen, even though it is a little heavy handed to use. LightMock required us to create separate objects that implements the interface we want to mock and pass calls to the LightMock framework.

---

```
public class MockBrandingRepository : IBrandingRepository
{
    private readonly IInvocationContext<IBrandingRepository>
        ↪ _context;

    public
        ↪ MockBrandingRepository(IInvocationContext<IBrandingRepository>
        ↪ context)
    {
        _context = context;
    }

    public void Add(BrandingItem item)
    {
        _context.Invoke(m => m.Add(item));
    }

    public BrandingItem Find(int Id)
    {
        return _context.Invoke(m => m.Find(Id));
    }
}
```

---

Finally, we needed to implement the logger in our tests. To do this we chose to simply use a stub [36], as the logger doesn't provide any data or processing needed for testing purposes.

### 5.1.1 Unit test example

Next we're going to do a step by step overview on how our tests work using one of our tests as an example. For this example we will use a test method called `getItemWithIdOk()`, which requests a `BrandingItem` object with a specific id, in our case 1, from the `BrandingController`. The intended result of this test is to successfully retrieve a `BrandingItem`.

All tests are split into three parts: arrange, act and assert. Firstly the test uses a `[Fact]` attribute to indicate to Xunit that the following method is a test. Next is the arrange part of the test, here we will create any values needed throughout the test as well as arranging the mocked repository. Here we have created a `BrandingItem` object we want to attempt to fetch, with some properties we can use to identify it with later. Then we use an arrange method on the repository which tells it to return the object we created if it's provided with the given `itemId` value. Next we need to set the Action Context of the controller so that we are able to test the status code [37] response. Lastly we add a cookie to the controller request which the controller extracts during execution to identify the user.

---

```
[Fact]
public void getItemWithIdOK()
{
    // Arrange
    var itemId = 1;
    var entity = new BrandingItem
    {
        OwnerId = itemId,
        Logo = "",
        PrimaryColor = "#3333",
        BrandName = "Consignor",
        BrandURL = "www.Consignor.com",
    };

    _repository.Arrange(m => m.Find(itemId)).Returns(entity);

    //Act: Set HttpContext and Valid Cookie
    _controller.ActionContext = new ActionContext
    {
        HttpContext = new DefaultHttpContext()
    };

    Dictionary<string, StringValues> dictionary = new
        ↳ Dictionary<string, StringValues>();
    string [] arr = { itemId.ToString() };
    dictionary.Add("installationID", arr);
    _controller.ActionContext.HttpContext.Request.Cookies = new
        ↳ ReadableStringCollection(dictionary);
}
```

---

}

---

In the act stage we simply call the method we are testing with any parameters needed. In this specific test the data received by the controller are located in the cookie we created during the arrange stage.

---

```

...
//Act:
ActionResult actionResult = _controller.Get();
...

```

---

Finally, we can make asserts to ensure that the final result is as expected. First we assert that the controller sent the appropriate status code. In this example the status code was 200, which indicates that the call was successful, and hence the response should contain the object we attempted to retrieve. It is important we do this before we attempt to extract the object from the response, because if the status code was not 200 the object would not be present in the response, and we would have gotten an unwanted error. Further, we extract the BrandingItem and make assertions on it's properties to make sure that it is in fact the object we wanted to get. Finally, we make an assertion on the repository which checks if the given repository method, in this case Find(), was called during the test.

---

```

...
//Assert
Assert.Equal((int)HttpStatusCode.OK,
    ↪ _controller.Response.StatusCode);

ObjectResult ob = (ObjectResult)actionResult;
BrandingItem item = (BrandingItem)ob.Value;

//Assert
Assert.Equal(itemId, item.OwnerId);
Assert.Equal("", item.Logo);
Assert.Equal("#3333", item.PrimaryColor);
Assert.Equal("Consignor", item.BrandName);
Assert.Equal("www.Consignor.com", item.BrandURL);

_repository.Assert(m => m.Find(itemId));

```

---

## 5.2 Component & System Testing

Component testing on communication between the front-end and back-end was done at the end of every sprint to ensure new functionality worked as intended. In addition regression testing on previously developed components was performed to ensure they were unaffected by the changes made to the system. To do this Fiddler [38] was used along with the chrome development tools [39]. Fiddler was especially helpful as it allowed us to inspect the API calls, providing us with information such as status code result, head-

ers and the response body. This made it easy to identify on which platform any errors occurred as we had complete track of all data sent between the two.

System testing was conducted less frequently than component testing, and was primarily performed at milestone events at which larger parts of the system had been completed. To conduct these test we set up a testing environment, how this was done is discussed in chapter 6, running the latest build, and manually tested the system. Performing these test was extremely helpful, as testing on a live environment revealed a lot of bugs that were not previously visible. This environment was also used periodically throughout development to test functionality that we were unable to fully test without a live environment, such as generating and sending emails and file storage.

### 5.3 Acceptance Testing

To perform the acceptance tests we made a mock interface which was used to imitate calls from Consignor to MGiNX. Through this interface we could create JSON payloads to simulate shipment packages, given to us by Consignor for testing purposes, to generate and send emails. These packages were made easily modifiable with the addition of the left side text boxes, used alter the content of the JSON package, which made for quick and easy testing of edge cases and different language configurations. After the JSON has been configured it is sent to the email engine by pressing the **Compile** button, which generates and sends the email to the receiver provided in the JSON. How this email should look can be previewed by clicking the **Display** button, which allows us to compare the intended result to how it is ultimately rendered by the different email readers. Finally the **Open Survey** button is there to generate a valid survey key and open up the survey webpage in a separate window.

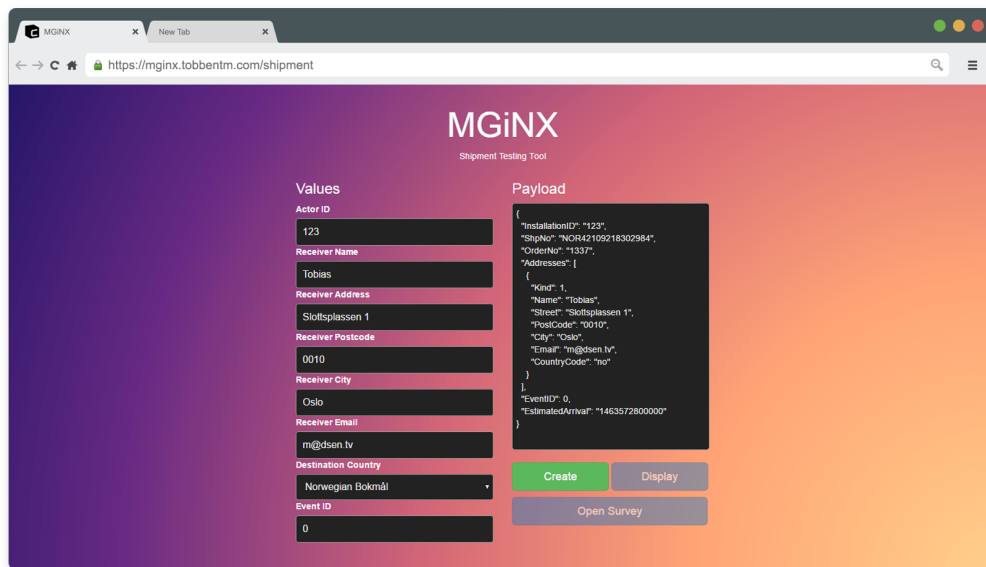


Figure 24: Screenshot of testing tool for emulating shipments.

### 5.3.1 Compliance Testing

We performed extensive compatibility testing of commonly used email readers and browsers. It was most important that we supported as many email readers as possible, as we have no control over which email readers receivers use. This was not that big of an issue with browsers as we could enforce browser requirements for actors, this was however not something we wanted to do.

As part of the compliance testing, we also tested our email templates in different email readers, to ensure consistent rendering and delivery. We tested everything from Microsoft Outlook 2003 to the more modern Google Gmail and Mozilla Thunderbird. During testing we found a small number of irregularities, and most of them were fixed by tweaking the templates. For example, Yahoo Mail would alter the CSS tag 'height' to 'min-height'. This made our padding elements have height = 0, because they had no real content. The solution was adding a non-breaking space using the HTML character "&nbsp;" to all padding elements. A complete list of tests can be found in [appendix I](#).

## 5.4 Swagger

In accordance with the documentation requirements we set in the planning phase, we have achieved complete coverage of the back-end. However, after having worked with an API for a few months we discovered further documentation was needed. Even though documentation of individual endpoints help from a developer standpoint it was still quite tricky, especially for consumers, to keep track of every endpoint.

To solve this problem we used the documentation tool Swagger UI [\[40\]](#). With Swagger UI we have created an interactive documentation experience aimed both towards developers and consumers alike. On the surface it offers a brief overview of all endpoints with their respective HTTP command, the full path and a short description. To gain more in-depth information every element can be expanded which reveals all required parameters and their location, most often either in the body or header of the request. Every endpoint also includes a list of all intended responses. These contain an explanation of what most likely caused that specific response, as well as a JSON schema of any complex objects included in the response. The interactive part of the API is the final part, which allows users to try out the endpoint by calling it with parameters filled in by the user. Below we have picked a single controller from our Swagger UI documentation to illustrate how this looks, for the complete version see [appendix O](#).



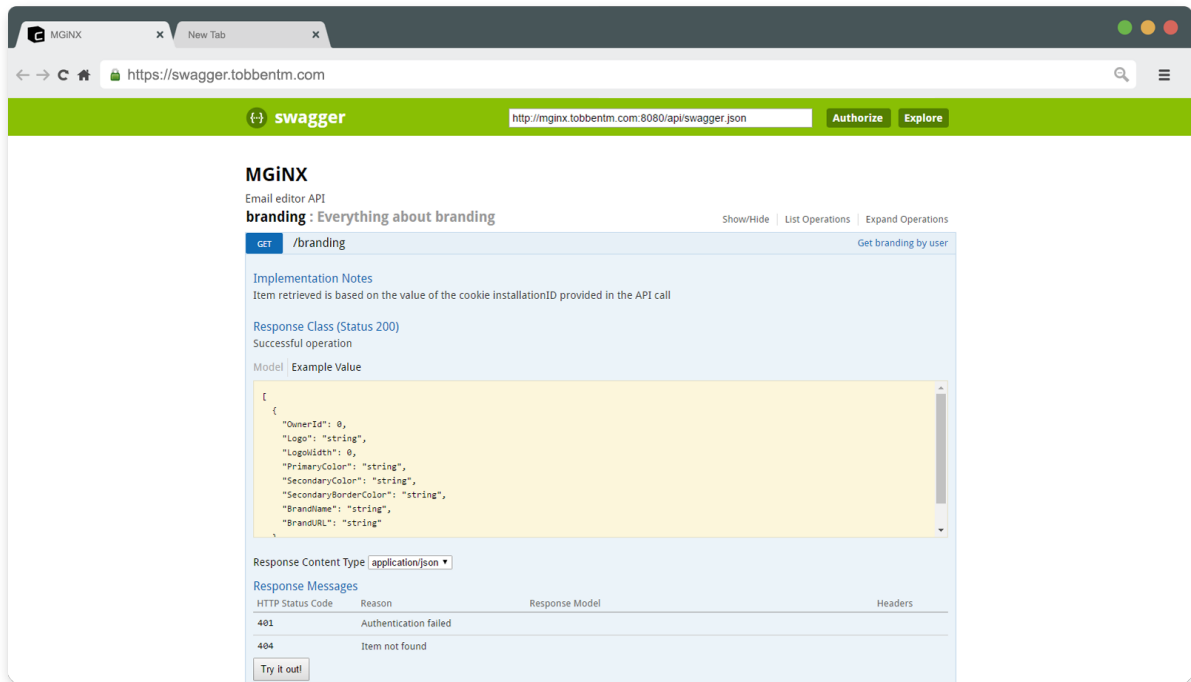


Figure 25: Swagger UI Documentation.

To achieve live testing using the tool, we needed to deal with CORS to allow for requests coming from the swagger domain (swagger.tobbentm.com) to be fulfilled by the backend. In ASP.Net we can do this during the application startup:

---

```
app.UseCors(builder =>
    builder.WithOrigins("http://swagger.tobbentm.com")
        .AllowAnyHeader()
        .AllowCredentials()
);
```

---

This allows the swagger origin, along with any headers and using cookie authentication. Any requests from other domains will not be allowed, to ensure user security and prevent CSRF attacks.

## 6 Deployment

### 6.1 Docker

New with .NET Core is that it can be run natively on Linux without the use of runtimes like Mono [41]. And to simplify building, deployment and devops, we chose to use Docker to run all necessary services. The following is a short description of each service we run in our test environment. The one service not inside a docker container is the Postfix server, more details about this in chapter 6.3. Certain containers are linked, may share volumes or have ports exposed.

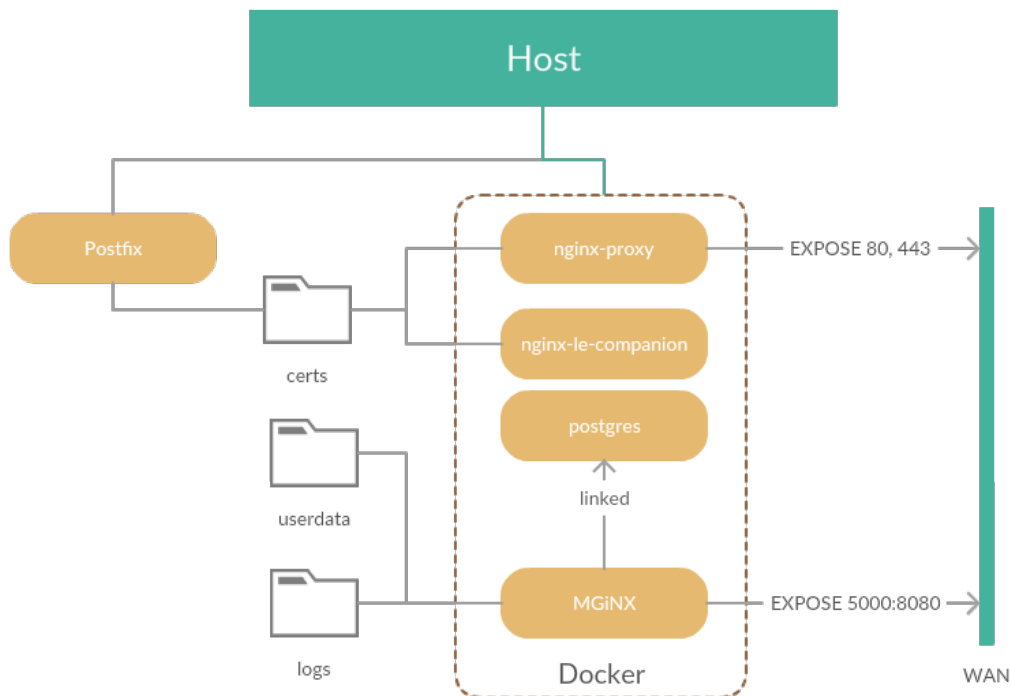


Figure 26: Diagram of host topology.

For this project we are hosting the application on a Debian based VPS host. Our test domain is 'mginx.tobbentm.com'.

#### 6.1.1 Postgres

Postgres is our choice of database for the project. For details on why we chose postgres, see chapter 4.3.

To run postgres, we have created a custom Dockerfile, which when installing runs a .sql script to add the necessary user to the database. We also copy in other necessary scripts

for populating the database with example data for required survey questions etc.

The database creation is handled by our ORM, Entity Framework 7, by compiling migrations and modifying the database. More detail about this in chapter 4.2.1. The postgres container is directly linked to by the MGiNX container, to allow direct communication.

### 6.1.2 nginx Proxy

Our application uses a .NET based web server called 'Kestrel', without support for SSL encryption. Because we wanted encryption as well as certificates we chose to run a proxy in front of our application, and landed on using nginx [1] as a proxy.

This also lets us easily cache static content, and force HTTPS. The Docker container we are using [42] have built in scripts to detect containers that need to be proxied, making the implementation very easy.

To flag a container for proxying we just have to add an environment variable:

---

```
.. -e VIRTUAL_HOST=mginx.tobbentm.com ..
```

---

Optionally we can also define a custom port if our container does not host on a 'default' port:

---

```
.. -e VIRTUAL_PORT=5000 ..
```

---

### 6.1.3 nginx Let's Encrypt Companion

As a result of wanting SSL encryption, we needed SSL certificates. To make it easy for ourselves, we chose Let's Encrypt [17], which delivers free and automated certificates. nginx Let's Encrypt Companion [43] automates this process in the same way as *nginx Proxy*, making the process of getting new certs very easy.

Like with *nginx Proxy*, this container looks for environment variables that define host name and contact information. In our case we are using the following flags:

---

```
.. -e LETSENCRYPT_HOST=mginx.tobbentm.com \  
-e LETSENCRYPT_EMAIL=webmaster@mginx.tobbentm.com ..
```

---

Once it detects new containers running with the flags, it will request a certificate from the CA, and store it in a folder shared with the host and the *nginx Proxy* container. On our server we store all certificates in '/var/mginx/certs'. This lets the *nginx Proxy* use them, as well as the Postfix email relay server.

### 6.1.4 MGiNX

The docker container for the application itself is based on the official image from Microsoft: 'microsoft/aspnet:1.0.0-rc1-final-coreclr'. Our Dockerfile is relatively simple, the steps are explained with comments:

```

# What image should we base the container on
FROM microsoft/aspnet:1.0.0-rc1-final-coreclr

# Copy application source into /app
COPY . /app

# Copy patched system libraries in
COPY ./System.Native.so
    ↪ /opt/DNX_BRANCH/runtimes/dnx-coreclr-linux-x64.1.0.0-rc1-update1/bin
COPY ./System.Native.so
    ↪ /opt/DNX_BRANCH/runtimes/dnx-coreclr-linux-x64.1.0.0-rc1-final/bin

# Change directory to /app
WORKDIR /app

# Restore NuGet libraries
RUN ["dnu", "restore"]

# Expose port 5000
EXPOSE 5000

# Define entrypoint to be dnx
ENTRYPOINT ["dnx", "-p", "project.json", "web"]

```

---

The patched system libraries are necessary because of a bug [23] in .NET Core RC1, as discussed in chapter 4.2.2.

The container can then be run with the following options:

---

```

alias hostip="ip addr show eth0 | grep -Eo -m 1 'inet
    ↪ (([0-9]{1,3}\.){3}[0-9]{1,3})' | awk '{print \$2}'"

docker run -d \
  --name mginx \
  -v /var/mginx/logs:/app/logging \
  -v /var/mginx/userdata:/app/userdata \
  -e VIRTUAL_HOST=mginx.tobtentm.com \
  -e VIRTUAL_PORT=5000 \
  -e LETSENCRYPT_HOST=mginx.tobtentm.com \
  -e LETSENCRYPT_EMAIL=webmaster@mginx.tobtentm.com \
  --add-host=dockerhost:$(hostip) \
  --link postgres-server:postgres \
  mginx

```

---

The '-v' options sets shared volumes, as illustrated in figure 26. The '-e' options sets environment variables necessary for other containers, as explained in chapter 6.1.2 and 6.1.3. The '-add-host' option will add a dns pointer in the container, in which we set the host 'dockerhost' to be the Docker bridge gateway. This way the application will have a reference to the host machine. This is primarily used for sending emails through. The '-link' option will link the container to the database container, and thus also register it as the

host 'postgres' withing the MGiNX container. During testing we also have an option to expose port 5000 in the container to port 8080 on the host. This lets us bypass the web proxy, as .NET Core RC1 had some bugs related to encryption of API calls. More details on this in chapter 4.2.

### 6.1.5 Swagger UI

As described in chapter 5.4, we are using Swagger to document our API interface, and Swagger UI to display it. This is easily done by using the official swagger distribution [44], and hosting it using a apache container. Following the patterns for our other containers, we can easily hook this into the web proxy, to be able to route it. We have set up a subdomain for the documentation: "swagger.tobbentm.com". To run the container, we have a handy one-liner:

---

```
docker run -it --rm --name swagger-ui -e
  ↪ VIRTUAL_HOST=swagger.tobbentm.com -v
  ↪ swagger-ui/dist /usr/local/apache2/htdocs/ httpd:2.4
```

---

## 6.2 Encryption

As explained in chapter 6.1.3, the *nginx Let's Encrypt Companion* container will automate the process of renewing SSL certificates, and will place them in the shared volume '/var/mginx/certs'. This is used by both the web proxy, to deliver an encrypted version of the application, and in our case also forces HTTPS over HTTP. The Postfix server will also use these certificates to encrypt all outgoing email with STARTTLS.

## 6.3 Postfix

To encrypt outgoing emails we opted to set up an email relay. This also lets us receive emails and manage everything email related more effectively, and improves the overall scalability of the system, as this can be offloaded to another server. We ended up choosing Postfix as the solution to this since Postfix is a relatively easy service to set up and configure. In our case it could be configured to relay email from Docker containers as local mail, and encrypt to remote recipients.

The way we have configured Postfix is by defining the following in the Postfix config file 'main.cf':

---

```
...

# MGiNX Settings
mydomain = tobbentm.com
myhostname = mginx.tobbentm.com
mydestination = $myhostname, localhost.$mydomain, localhost,
  ↪ $mydomain
mynetworks = 127.0.0.1/8 [::1]/128 172.16.0.0/12
virtual_alias_domains = mginx.tobbentm.com
virtual_alias_maps = hash:/etc/postfix/virtual

smtpd_use_tls = yes
smtpd_tls_security_level = may
```

```
smtpd_recipient_restrictions = permit_mynetworks
    ↪ reject_unauth_destination
smtpd_tls_key_file = /var/mginx/certs/mginx.tobbentm.com/key.pem
smtpd_tls_cert_file =
    ↪ /var/mginx/certs/mginx.tobbentm.com/fullchain.pem
smtpd_tls_loglevel = 1

smtp_tls_security_level = may
smtp_tls_loglevel = 1
```

---

The 'mynetworks' option defines which networks should be considered as local. By adding the 172.16.0.0/12 subnet, we are defining our Docker bridge subnet as being a local network. This effectively means any mail coming from an address within the subnet does not need to authenticate with the relay, and Postfix will simply relay the email to the recipient.

By setting 'smtpd\_tls' and 'smtp\_tls' options, we are merely enabling STARTTLS encryption, and using the certificates shared amongst the containers mentioned in [chapter 6.2](#).

For testing purposes we have set the virtual mapping of postboxes to send all mail to the root unix mailbox using the follow virtual map:

---

```
@mginx.tobbentm.com root
```

---

## 7 Conclusion

### 7.1 Results

Of the requirements made during project planning we were able to complete most of them. Overall we are very pleased with the result, and so was the product owner during the last demo and sprint meeting. We have managed to create an easily extensible platform for managing email configuration and delivery. It lets the actors easily modify email appearance and content, send out surveys to receivers and get real feedback in the form of reports. All while delivering the email in the most trustworthy way possible using SPF, DKIM, encryption and more, with almost guaranteed compatibility with the most popular email clients. And by using ASP.Net Core, we have a solid foundation for integrating with other Consignor systems and to push the limits of what is possible in terms of performance and scalability.

The functionality we did not get to implement, was largely due to technical issues (see chapter 4.5.2). The challenges of working with prototype software became very apparent as we needed more and more from the ASP.Net framework further into the project. In the end though, we have learned a lot about potential issues, workarounds and how to trace and fix these kinds of issues.

Due to time constraints we were unable to implement certain features. Sending surveys through SMS, allowing actors to use their own email server and the ability to order survey answers by a multitude of shipment specific variables such as carrier, location and date are all examples of this. In addition, due to the fact that we were using Scrum it meant that some features were introduced late into development, and as such we did not have sufficient time to develop them. One such feature was providing context to survey answers through shipment data. Consignor identified that this could be very valuable information to the end-users, however it was simply introduced too late into development.

Having bi-weekly sprint meetings with the product owner has also been very helpful for both parties. Consignor in general have been very supportive, and have sponsored trips to their headquarters in Oslo for more important meetings and project demonstrations. They were always very open to new ideas, allowing us to implement much of the functionality we wanted as well.

### 7.2 What would we do differently today?

When we first started planning the implementation we were thinking that the editor was going to be a large part of the project. We expected that most of our client-side code would probably be vanilla JavaScript, hence we didn't need a front-end framework. As the project went on the amount of pages, dialogs and navigation options grew, and we would probably have been better off learning and using a front-end framework.

We should have started writing the report at a much earlier stage. From the beginning we planned to start writing the report mid-way through development, in addition to dedicating the final 4 weeks to finish the report. However, as development went on adding new functionality got prioritized over writing the report, ultimately delaying report writing completely to the final 4 weeks. We realized this error too late, and a lot of extra time has been spent the last few weeks to compensate for that.

### 7.3 Further Development

There is almost infinite possibility with this project, and most importantly; it's built to be extended and improved. The following list highlights some of our most wanted functionality:

- **Reports**
  - Ability to filter data based on shipment data (geographically, carrier based, etc).
- **Email Editor**
  - Better handling of iterations, ability to browse older iterations than the ones displayed in the editor.
  - Indicator for whether or not you have unsaved changes to keep user from accidentally undoing changes.
  - Refactor out editor specific javascript to a separate object for better clarity and handling.
- **Settings**
  - Allow user to define custom email relay or custom sender email address (technical limitation at the moment).
- **Branding**
  - More helpful text for the user as to how the data is actually used when sending emails.
- **Email Engine**
  - Implement a queue to ensure deliverability to all shipments. Important if there is a lot of incoming shipments at the same time.
  - Solve the problem of custom email as described in chapter [4.5.2](#).
- **General/Misc**
  - Better handling of Docker container building and running (installation).

While none of this is strictly necessary for a bare-bones instance, it would certainly improve the user experience and versatility of the application.

### 7.4 Group Evaluation

We had not worked together prior to this projects and we had barely spoken to each other, so we both started this semester with light skepticism, not fully knowing what to expect of the other. Ultimately, however, we were both pleasantly surprised by the others



determination and commitment to the project.

The teamwork in the group has been amazing, we have both been involved in almost every part of the application, yet maintained a clear focus on our own tasks. We both contributed differently in regards to project management which we think have worked out pretty well. The amount of hours we put into this project was pretty close to the plan, with both of us spending roughly 30 hours per week on research, development or testing. Each sprint was fairly on schedule and tasks got done in time, which made things easier to estimate as the project progressed.

As mentioned in chapter 1 we divided the administrative tasks evenly, as well as having each team member specialize in different fields during development. By doing so no member got overwhelmed with responsibilities which allowed us to focus on researching and learning our respective fields. Due to us working together most of the day this meant that as our work intertwined the other would always be available to answer any question, which resulted in a more efficient work-flow. To begin with we didn't take advantage of this as we didn't know each other that well. However, after having worked together all day for a couple of weeks that quickly passed. Working with an API made it a lot easier to distribute our work in this manner, as using the API endpoints didn't require detailed knowledge of the codebase behind the controllers, just its inputs and outputs.

There have been no major conflicts or disagreements within the group, and having such different backgrounds, each with his own experience, we have been able to share and help each other throughout the project.

## **7.5 Conclusion**

In conclusion, MGiNX provides a prototype platform for managing email delivery in a very versatile way. With a powerful editor, both for non-technical users as well as technical users, and a solid email generation process, the platform delivers industry grade email delivery and customization options. While we have some minor regrets and ran into some unnecessary bumps in the road, we are very pleased with the project as a whole, both in terms of functionality implemented and teamwork/management. We have hopes that this will be put into production some time in the future, and that our work actually sees real world use.

## Bibliography

- [1] nginx. 2016. nginx, about. <http://nginx.org/en/>. (Visited Apr. 2016).
- [2] Microsoft. 2015. Introducing .net core. <http://docs.asp.net/en/latest/conceptual-overview/dotnetcore.html>. (Visited May. 2016).
- [3] Shackel§, B. 1986. Ibm makes usability as important as functionality. *The Computer Journal*, 29(5), 475–476.
- [4] Nielsen, J. 2002. Usability 101: Introduction to usability. <https://www.nngroup.com/articles/usability-101-introduction-to-usability/>. (Visited May. 2016).
- [5] W3Schools. 2016. Browser statistics. [http://www.w3schools.com/browsers/browsers\\_stats.asp](http://www.w3schools.com/browsers/browsers_stats.asp). (Visited May. 2016).
- [6] litmus. 2015. 53 <https://litmus.com/blog/53-of-emails-opened-on-mobile-outlook-opens-decrease-33>. (Visited May. 2016).
- [7] Microsoft. 2015. Xml documentation comments (c# programming guide). <https://msdn.microsoft.com/en-us/library/b2s063f7.aspx>. (Visited May. 2016).
- [8] Microsoft. 2015. Sandcastle - documentation compiler for managed class libraries. <https://sandcastle.codeplex.com/>. (Visited May. 2016).
- [9] Dustin, E. 2002. *Effective Software Testing: 50 Specific Ways to Improve Your Testing*. Addison-Wesley Professional.
- [10] Serilog. 2015. Flexible, structured events — log file convenience. <http://serilog.net/>. (Visited May. 2016).
- [11] Teamwork.com. 2016. High performance teams run on teamwork. <https://www.teamwork.com/>. (Visited May. 2016).
- [12] GitHub. 2016. How people build software. <https://github.com/>. (Visited May. 2016).
- [13] IETF. 2014. Sender policy framework (spf) for authorizing use of domains in email, version 1. <https://tools.ietf.org/html/rfc7208>. (Visited Apr. 2016).
- [14] IETF. 2011. Domainkeys identified mail (dkim) signatures. <https://tools.ietf.org/html/rfc6376>. (Visited Apr. 2016).
- [15] Google. 2016. Email encryption in transit. <https://www.google.com/transparencyreport/saferemail/>. (Visited May. 2016).

- [16] IETF. 2002. Smtplib service extension for secure smtp over transport layer security. <https://tools.ietf.org/html/rfc3207>. (Visited Apr. 2016).
- [17] (ISRG), I. S. R. G. 2016. Let's encrypt. <https://letsencrypt.org/>. (Visited Apr. 2016).
- [18] Foundation, N. 2016. Nuget. <https://www.nuget.org/>. (Visited May. 2016).
- [19] Bower. 2016. Bower, a package manager for the web. <http://bower.io/>. (Visited May. 2016).
- [20] Git. 2016. Git. <https://git-scm.com/>. (Visited May. 2016).
- [21] Npgsql. 2015. .net data provider for postgresql. <http://www.npgsql.org/>. (Visited Apr. 2016).
- [22] PostgreSQL. 2016. The world's most advanced open source database. <http://www.postgresql.org/>. (Visited Apr. 2016).
- [23] Network, M. D. 2015. Socket.select() method doesn't work correctly in linux. <https://github.com/dotnet/corefx/issues/4631>. (Visited Apr. 2016).
- [24] halter73. 2015. Don't wait to consume the entire request body for connection: close requests. <https://github.com/aspnet/KestrelHttpServer/issues/406>. (Visited Apr. 2016).
- [25] Microsoft. 2016. Fluent api. <http://ef.readthedocs.io/en/latest/modeling/keys.html#fluent-api>. (Visited May. 2016).
- [26] Twitter. 2016. Get bootstrap. <http://getbootstrap.com/>. (Visited May. 2016).
- [27] jQuery Foundation, T. 2016. jquery. <https://jquery.com/>. (Visited May. 2016).
- [28] Consignor. 2016. Consignor portal. <http://www.consignorportal.com/>. (Visited Apr. 2016).
- [29] Network, M. D. 2015. Content editable. [https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content\\_Editable](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_Editable). (Visited May. 2016).
- [30] jakiestfu. 2016. Medium.js. <https://github.com/jakiestfu/Medium.js/>. (Visited May. 2016).
- [31] Stedfast, J. 2015. Mimekit. <http://www.mimekit.net/>. (Visited May. 2016).
- [32] IETF. 2001. Internet message format. <https://tools.ietf.org/html/rfc2822>. (Visited Apr. 2016).
- [33] Stedfast, J. 2016. Mailkit. <https://github.com/jstedfast/MailKit>. (Visited May. 2016).
- [34] Foundation, O. 2015. About xunit.net. <https://xunit.github.io/>. (Visited May. 2016).
- [35] Richter, B. 2015. Lightmock. <https://github.com/seesharper/LightMock>. (Visited May. 2016).

- [36] Wikipedia. 2016. Test stub. [https://en.wikipedia.org/w/index.php?title=Test\\_stub&oldid=671253434](https://en.wikipedia.org/w/index.php?title=Test_stub&oldid=671253434). (Visited May. 2016).
- [37] Wikipedia. 2016. List of http status codes. [https://en.wikipedia.org/w/index.php?title=List\\_of\\_HTTP\\_status\\_codes&oldid=719266042](https://en.wikipedia.org/w/index.php?title=List_of_HTTP_status_codes&oldid=719266042). (Visited May. 2016).
- [38] Telerik. 2016. Fiddler the free web debugging proxy for any browser, system or platform. <http://www.telerik.com/fiddler>. (Visited May. 2016).
- [39] Google. 2016. Chrome devtools overview. <https://developers.google.com/web/tools/chrome-devtools/>. (Visited May. 2016).
- [40] SmartBear. 2016. Swagger the world's most popular framework for apis. <http://swagger.io/swagger-ui/>. (Visited May. 2016).
- [41] Project, M. 2016. Mono project. <http://www.mono-project.com/>. (Visited Apr. 2016).
- [42] jwilder. 2016. Automated nginx proxy for docker containers using docker-gen. <https://github.com/jwilder/nginx-proxy>. (Visited Apr. 2016).
- [43] JrCs. 2016. Letsencrypt companion container for nginx-proxy. <https://github.com/JrCs/docker-letsencrypt-nginx-proxy-companion>. (Visited Apr. 2016).
- [44] SmartBear. 2016. Swagger ui. <https://github.com/swagger-api/swagger-ui>. (Visited May. 2016).



## Appendix



## Terminology

**Actor and Sender** Refers to end-users of the final product. These are clients of Consignor who are using their software. The use of actor or sender depends on the context. Actor refers to a user in context to using the software, whereas sender is in context to a user during a transaction.

**Customer and Receiver** Both Receiver and Customer refer to the same people, but in different context. Customer refers to people who are using the actors online shopping system, whereas receivers are currently part of a transaction with the actor.

**Shipment** Refers to the physical delivery of a package currently being sent to a customer.

**Shipment Data** A JSON object used in MGiNX that contains all data relevant to a package and its receiver.

**Carrier** Companies used to transport packages from sender to receiver.

**CORS** Cross-origin resource sharing

**CSFR** Cross-site request forgery

**DAL** Data Access Layer (models/repository)

**RC(#)** Release Candidate; early/unrelease version of software

**MTA** Mail Transfer Agent; used to relay email

**MUA** Mail User Agent; used to send and receive email

**SPF** Sender Policy Framework; see page [20](#)

**DKIM** DomainKeys Identified Mail; see page [20](#)

**Postgres** Database software; see page [29](#)



## Appendix

# ❧ B ❧

## Project Description

**Oppdragsgiver:** EDI-SOFT AS

**Kontaktpersoner:** Øystein Ranvik / Bjørn Pedersen

**Adresse:** Rådhusgata 4, 0151 Oslo

**Telefon:** +47 93 49 60 75  
+47 91 71 54 52

**Epost:** [or@edi-soft.no](mailto:or@edi-soft.no)  
bjorn@edi-soft.no

## Om EDI-Soft

EDI-Soft leverer løsninger for å administrere forsendelser fra bedrifter. Løsningen er ofte integrert med avsenders økonomi-/ordre system og hjelper avsender med å skrive ut nødvendige fraktdokumenter, beregne fraktpris, informere transportør, håndtere sporing osv.

## Bakgrunn

En stadig større andel av varehandel skjer via netthandel. Bedriftene (avsender) bruker store ressurser på å gjøre handleopplevelsen i nettbutikken best mulig (fylle handlekurven + betaling). Her finnes det mange gode løsninger i dag.

Selve leveransen håndteres gjerne av eksterne transportører (majoritet), utlevering på pickup points eller i egne butikker. Når transportøren henter pakkene hos avsender er salget ferdig for avsender, men for mottaker er selve leveransen en betydelig del av handleopplevelsen.

Eksempel: Kjøp av Apple TV. Produktet er likt, prisen er lik, nettbutikken er like god. Årsaken til at Komplette velges fremfor Elkjøp, kan i mange tilfeller skyldes selve leveranseprosessen.

Avsender har i dag svært liten kontroll på leveranseopplevelsen til mottaker.

## Oppgaven

Utvikle et system for kommunikasjon fra avsender til mottager og tilbakemeldinger fra mottager til avsender.

Oppgaven er firedelt hvor følgende ting skal designes og utvikles.

1. En editor hvor brukeren kan definere hvordan eposter som blir sendt til kunder skal se ut og hvilke informasjon den skal inneholde. Dette må lagres på et valgt format slik at det kan leses



inn i editoren for videre redigering. Web GUI, mulig å legge til vedlegg, bruk av tokens eller fast tekstog grafikk.

2. En motor som generer en epost fra data om sendingen og det som brukeren har definert i editoren. Når spesielle hendelser inntreffer vil denne modulen bli kalt. Her vil det være med data for å finne frem til riktig konfigurering. Dataene for feltene som var tilgjengelig i editoren vil bli sendt inn som xml/json. Dette bør implementeres som en webservice.
3. Lage en løsning (konfigurator og landingsside) hvor mottagere av sendinger kan gå inn og gi tilbakemelding på kvaliteten på leveransen eller andre ting som avsender ønsker. Dette skal være konfigurerbart for den enkelte avsender. Link til denne side blir sendt i epost/sms. Det er viktig at denne er mobiltilpasset. Det må også være mulig å bruke templates (EDI-Soft default)
4. Dashboard, som viser utvalgte data  
Lage rapporter på tilbakemeldinger fra kunder, her skal det være mulig å se både statistikk og de enkelte tilbakemeldingene. Filtrering på transportør, geografi ++

Oppgave vil passe for 2 personer.



## Appendix

❧ C ❧

## Project Agreement



HØGSKOLEN I GJØVIK

## PROSJEKTAVTALE

mellom Høgskolen i Gjøvik (HiG) (utdanningsinstitusjon),

CONSIGNOR GROUP AS

(oppdragsgiver), og

Tobias Lønnerød Madsen,

Thomas Sem - Jacobsen

(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 01.01.2016 til 16.05.2016.

Studentene skal i denne perioden følge en oppsatt fremdriftsplan der HiG 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 HiG 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 HiG. Studentene dekker utgifter for trykking og ferdigstillelse av den skriftlige besvarelsen vedrørende prosjektet.
  - 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. HiG 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 faglærer/veileder og sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.
4. Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode, disketter, taper mv. som inngår som del av eller vedlegg til besvarelsen, gis det en kopi av til HiG, som vederlagsfritt kan benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av HiG 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 HiG og/eller studenter har interesser.

Besvarelser med karakter C eller bedre registreres og plasseres i skolens bibliotek. Det legges også ut en elektronisk prosjektbesvarelse uten vedlegg på bibliotekets del av skolens internett-sider. Dette avhenger av at studentene skriver under på en egen avtale hvor de gir biblioteket tillatelse til at deres hovedprosjekt blir gjort tilgjengelig i papir og netttutgave (jfr. Lov om opphavsrett). Oppdragsgiver og veileder godtar slik

offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og dekan om de i løpet av prosjektet endrer syn på slik offentliggjøring.

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 Fronter. I tillegg leveres et eksemplar til oppdragsgiver.
8. Denne avtalen utferdiges med et eksemplar til hver av partene. På vegne av HiG er det dekan/prodekan som godkjenner avtalen.
9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og HiG som nærmere regulerer 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, skjer dette uten HiG som partner.

10. Når HiG også opptrer som oppdragsgiver trer HiG inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.
11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene i mellom. 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.

12. Deltakende personer ved prosjektgjennomføringen:

HiGs veileder (navn):

Frode Haug

Oppdragsgivers  
kontaktperson (navn):

Bjørn PEDERSEN / ØVSTEIN RANVIK

Student(er) (signatur):

[Signature]  
[Signature]

dato 14/02-16

dato 14/01-16

dato \_\_\_\_\_

dato \_\_\_\_\_

Oppdragsgiver (signatur):

[Signature] [Signature] dato 14/01-16

IMT Dekan/prodekan (signatur):

dato \_\_\_\_\_



## Appendix

# ❧ D ❧

## Group Rules

# Group Rules

- Hvis gruppen er uenige har gruppe leder siste ordet, med mindre de andre gruppe medlemene mener det er en sak viktig nok for at veileder skal tas inn. I et slikt tilfelle skal veileder veilede diskusjonen i gruppen, og hvis gruppen ikke kan komme til enighet har gruppeleder fremdeles siste ordet.
- Hvis et gruppemedlem ved gjentatte ganger ikke leverer tildelt arbeid, og ikke selv har tatt opp saken med gruppen, skal dette meldes til veileder og en advarsel skal gis til gruppemedlemet. Ved tredje varsel har veileder med samtykke av resten av gruppen fullmakt til å splitte gruppen.
- Hvis gruppen splittes skal alle ferdige sprint elementer for nåværende sprint pushes til git repository.
- Ved fravær fra faste møtetider og avtalte møter med veileder eller Consignor skal dette meldes fra minst 2 dager på forhånd.
- Kostnader knyttet til prosjektet som Consignor selv ikke velger å ta på seg, skal deles likt mellom gruppemedlemene.
- Retningslinjer for kode standard, dokumentasjon og testing skal følges, hvis dette ikke er gjennomført ses sprint elementet som ikke gjennomført.

---

---



## Appendix

❧ **E** ❧

## Hour Log

Date	Team member	Hours	Activity
12.01	Tobias	5	Project Planning
12.01	Tobias	2	Project Website
12.01	Thomas	6	Project Planning
12.01	Thomas	2	Learning Technologies
13.01	Tobias	2	Project Website
13.01	Tobias	2	Wireframe Mockup
13.01	Tobias	1	Project Planning
13.01	All Members	2	Research
13.01	Thomas	7	Project Planning
14.01	All Members	9	Meeting in Oslo with Consignor/Product Owner
15.01	Tobias	1	Meeting Summary
15.01	Tobias	1	Project Website
15.01	Tobias	3	Project Planning
15.01	Thomas	5	Project Planning
16.01	Thomas	5	Research
18.01	Tobias	7	Project Planning
18.01	Thomas	5	Project Planning
18.01	Thomas	2	Research
19.01	All Members	1	Meeting with supervisor
19.01	Thomas	2	Project Planning
19.01	Thomas	3	Learning Technologies
19.01	Tobias	5	Project Planning
20.01	All Members	7	Project Planning (SRS)
21.01	All Members	2	Research
21.01	All Members	4	Project Planning (SRS)
22.01	All Members	1	Sprint Planning
22.01	All Members	2	Research
23.01	Thomas	2	Learning Technologies
24.01	Thomas	2	Research
24.01	Thomas	1	Setting up the project
25.01	All Members	7	Developing
26.01	All Members	.5	Meeting with supervisor
26.01	All Members	.5	Meeting with Consignor
26.01	All Members	6	Development
27.01	Tobias	7	Development
27.01	Thomas	4	Development
27.01	Thomas	4	Research
28.01	All Members	7	Development
29.01	All Members	3	Development
01.02	Tobias	7	Development
01.02	Thomas	5	Development
01.02	Thomas	2	Research



02.02	All Members	7	Development
02.01	All Members	.2	Meeting with supervisor
03.02	All Members	6	Development
04.02	All Members	6	Development
05.02	All Members	5	Development
08.02	All Members	.5	Sprint review and planning meeting
08.02	Tobias	6	Development
08.02	Thomas	2	Development
09.02	All Members	6	Development
09.02	All Members	.1	Meeting with supervisor
10.02	All Members	7	Development
11.02	All Members	7	Development
12.02	All Members	5	Development
15.02	All Members	7	Development
16.02	All Members	6	Development
16.02	All Members	.2	Meeting with supervisor
17.02	All Members	6	Development
18.02	All Members	7	Development
19.02	All Members	5	Development
22.02	All Members	7	Development
23.02	All Members	7	Development
23.02	Tobias	.2	Meeting with supervisor
24.02	All Members	7	Development
25.02	Tobias	7	Development
25.02	Thomas	5	Development
26.02	All Members	5	Development
29.02	Thomas	5	Development
01.03	All Members	6	Development
02.03	All Members	6	Development
03.03	All Members	6	Development
04.03	All Members	4	Development
07.03	All Members	5	Development
07.03	All Members	.5	Sprint review and planning meeting
08.03	All Members	6	Development
08.03	All Members	.2	Meeting with supervisor
09.03	All Members	5	Development
09.03	All Members	1	Bachelor lynkurs
10.03	All Members	6	Development
11.03	All Members	5	Development
14.03	All Members	7	Development
15.03	All Members	7	Development
16.03	All Members	6	Development
17.03	All Members	7	Development
18.03	All Members	.5	Sprint planning and review meeting with Con-signor
18.03	All Members	4	Development
21.03	Tobias	7	Development
22.03	All Members	6	Development
23.03	All Members	5	Development
29.03	All Members	.3	Meeting with supervisor
29.03	All Members	6	Development

30.03	All Members	6	Development
31.03	All Members	7	Development
01.04	All Members	5	Development
04.04	All Members	7	Development
05.04	All Members	7	Development
05.04	All Members	0.2	Meeting with supervisor
06.04	All Members	6	Development
07.04	All Members	11	Development and demo preparation
08.04	All Members	6	Demo of software for product owner
11.04	All Members	6	Development
12.04	All Members	7	Development
13.04	All Members	7	Development
14.04	All Members	6	Development
15.04	All Members	4	Development
18.04	All Members	6	Development
19.04	Tobias	6	Development
19.04	Tobias	0.2	Meeting with supervisor
19.04	Thomas	5	Report Structure
20.04	Tobias	6	Development
20.04	Thomas	5	Report Structure
21.04	Tobias	6	Development
21.04	Thomas	5	Report Structure
22.04	Tobias	4	Development
22.04	Thomas	4	Report Structure
25.04	Tobias	3	Development
25.04	Tobias	3	Report
25.04	Thomas	6	Report
25.04	All Members	0.5	Meeting with Consignor
26.04	Tobias	7	Report
26.04	Thomas	9	Report
27.04	All Members	6	Report
28.04	All Members	7	Report
29.04	All Members	5	Report
02.05	All Members	7	Report
03.05	All Members	7	Report
04.05	All Members	6	Report
04.05	All Members	0.5	Meeting with supervisor
05.05	All Members	6	Report
06.05	All Members	5	Report
09.05	All Members	7	Report
10.05	All Members	7	Report
11.05	All Members	9	Report
12.05	All Members	0.5	Meeting with supervisor
12.05	All Members	6	Report
13.05	All Members	6	Report
14.05	All Members	6	Report
15.05	All Members	7	Report
16.05	All Members	7	Report
17.05	All Members	5	Report



## Appendix



## Project Plan

# Consignor MGiNX

Bsc. Project Plan - 2016



Tobias Linnerd Madsen  
Thomas Sem-Jacobsen

# Project plan

## 1 Introduction

### 1.1 Background

Consignor (previously EDI-Soft) is a technology company headquartered in Oslo, who offer solutions to online retailers which store and manage shipment and package data. This solution integrates with the client's existing online ordering system, and assist in printing crucial shipment documents, calculating shipment costs for carriers in their area, alerting carriers of shipment events and tracking package shipment status. This gives retailers a greater control of the shipping companies available in their region and their relations with them. However, they still lack the ability to influence the user experience during the shipment stage.

### 1.2 Objectives

#### 1.2.1 Learning Objectives

For this project, we aim to gain knowledge in a number of fields and technologies. Some of them being:

- ASP.net Technologies with WebAPI and more
- Web technologies for creating responsive applications
- Scrum Software Methodology
- System Architecture using API driven web architecture
- Project planning and execution

#### 1.2.2 Impact Objectives

Provide Consignor with a complete piece of software that can be easily integrated with their existing system, and is designed to provide clients of Consignor with a system in which they can customize their user experience to suit their needs.

#### 1.2.3 Performance Objectives

Considering the growing market of logistics software and a need for customizability for users, we aim to fulfill the following performance objectives:

- Create a easily extensible and maintainable system
- Create a scalable and robust system

- Give more control to senders surrounding branding and communications with receivers of shipments
- Collect and display relevant statistics regarding communications with receivers and receiver happiness and satisfaction

### **1.3 Limitations**

- Our system is unaware of any data related to package transactions, and will therefore rely on Consignor's existing system to provide updates to a package shipment status.
- Our system is to provide a webpage in which the recipient can provide feedback on their experience with the sender and transactions between them. This feedback page will be provided through both e-mail and sms, and as per Consignor's request, this webpage is required to be responsive as to accommodate both stationary and mobile devices.
- We will not be doing any work to integrate with Consignors existing software. However, we will be using sample data and examples of configurations provided by Consignor to make the system as compliant as possible, for an easier integration at a later point.
- During development there will be no authentication needed to access the sender-side of the website. At a potential integration stage our system would have to authenticate with Consignors system.

### **1.4 Target Audience**

We have two separate audiences, one for the receiver-oriented email and survey components, and another for the dashboard, email editor and survey configurator components. The receiver is further split into two categories, businesses and the general public. Consignor is not limited to Norwegian customers, and we must therefore support internationalization in our project.

#### **1.4.1 Email & Survey**

The consumers for these components are shipment receivers. They are non-technical end-users, and often on small mobile devices. As such, the UI needs to be intuitive and responsive.

#### **1.4.2 Dashboard, Email Editor & Survey Configurator**

The consumers for these components are shipment senders, and are bound to desktop computers. They range from light users of the service, to power-users. The interface for these services therefore needs to be intuitive, as well as offering enough complex tools to allow for the desired configuration.

## **2 Scope**

### **2.1 Field of Study**

The majority of all transactions occur online, and so the shipment stage takes up the majority of the transaction. Retailers acknowledges that the user experience is a crucial part of the transaction, and hence a lot of resources have been used to improve the online shopping experience, and there exist a lot of good solutions here today. However, during the transaction there is a split between retailer and customer. For the retailer the transaction is finished once the transporter collects the package, but for the customer this is a vital part of the transaction, and this experience can be the deciding factor on the customers view of the retailer.

### **2.2 Technical Scope**

This is where Consignor see an opportunity to expand their current solution, by including a system which assist in maintaining the relationship between retailer and customer, even during shipment. This is achieved by allowing retailers to define the layout, content and design of emails sent to customers at specific shipment events. In addition, the system should offer customers a web interface in which they may provide feedback to retailers surrounding the shipment process, and any other questions specified by the retailer.

Combined this should give retailers a greater control of the user experience during shipment, and help narrow the split between retailer and customer.

### **2.3 Project Description**

The application will consist of 4 main components:

1. An email editor to easily customize email configurations
2. En email engine to inflate email configurations with relevant information
3. A survey editor to configure surveys to be sent to receivers
4. A dashboard to generate statistics and reports from surveys and historical data

Our suggested system architecture looks like this:

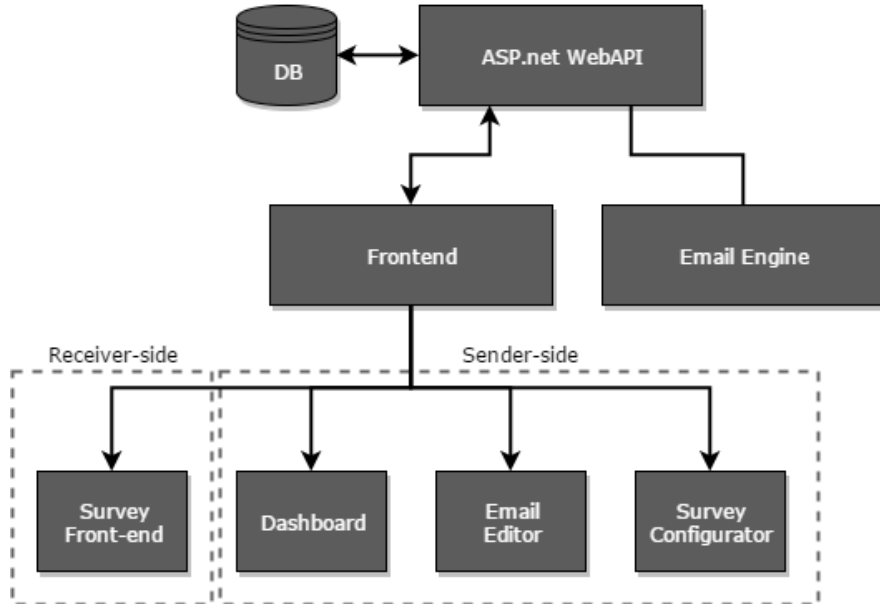


Figure 1: System Architecture

### 2.3.1 Email Editor

The email editor should be a easy way for users to either adapt preconfigured templates to their use cases, or to feed the engine premade HTML configurations.

The system should be able to keep track of several languages, with one configuration per language. When adding a new language, you should at all times be able to copy the text, images and layout from a different language configuration.

In addition to language, there may also be multiple events to respond to. The same requirements apply here.

When writing the email, the user should have easy access to variables. These variables will be injected with data when the email is sent. The variables available is based on the Consignor configuration. The configuration containing available variables will be received from Consignor during setup.

The following wireframe is our initial concept for the solution. The difference in implementing would mainly be a relocated and redesigned variable management, as well as relocating the 'iterations' column.



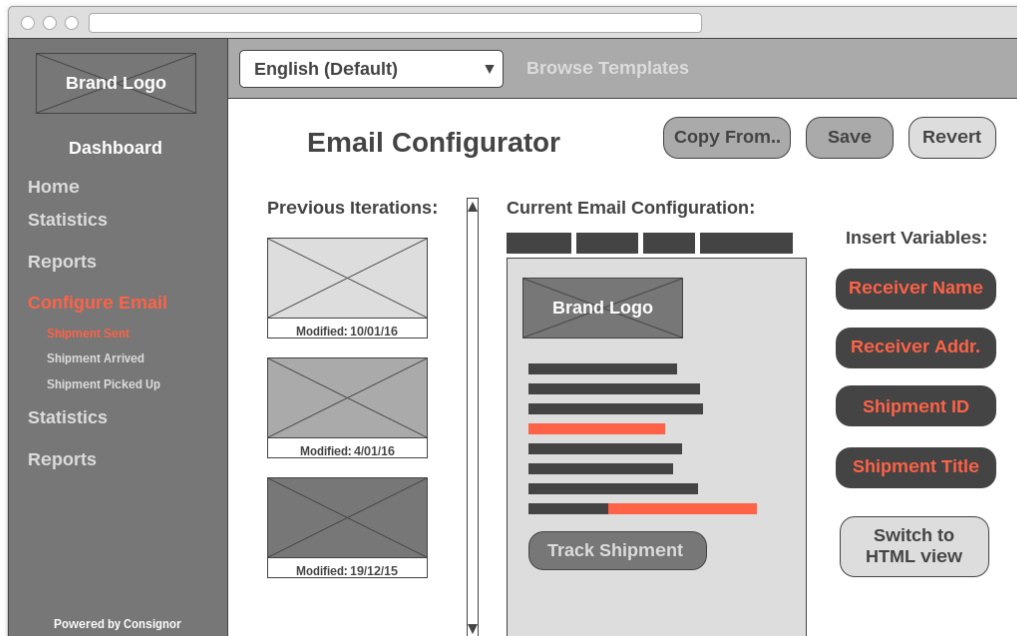


Figure 2: Wireframe for the email editor

### 2.3.2 Email Engine

The email engine is a webservice that will get notified by the Consignor system when a shipment has reached certain events. Based on the event, an email may be generated, using a customized configuration based on the actor, event and choosing the correct language based on destination country. The engine will take care of all details in create the email, including adding static or dynamic attachments, inflating variables with data and logging data.

The email engine should send emails on behalf of the user, and as such the system will need to get the required configuration from Consignor. To prevent the server in question from being blacklisted as spam, the system will occasionally have to check DNS records, to see if the required SPF records are up to date and allows the system to send emails on behalf of the user. If the SPF records are not found, an error message should appear. We may also have to fall back to using an in-house sender email to ensure service availability.

### 2.3.3 Survey Editor

Similarly to the email editor, this component lets senders define a survey to send to receivers after certain shipment events. The editor will let senders ask questions with a question text, and select a type of rating or answer.

The initial types of responses to questions is the following:

1. Star rating (1-5)
2. Happiness rating (1-5)
3. Text Answer
4. Custom Answer List

The list may be extended or altered during the project.

Our initial idea for the GUI is the following:

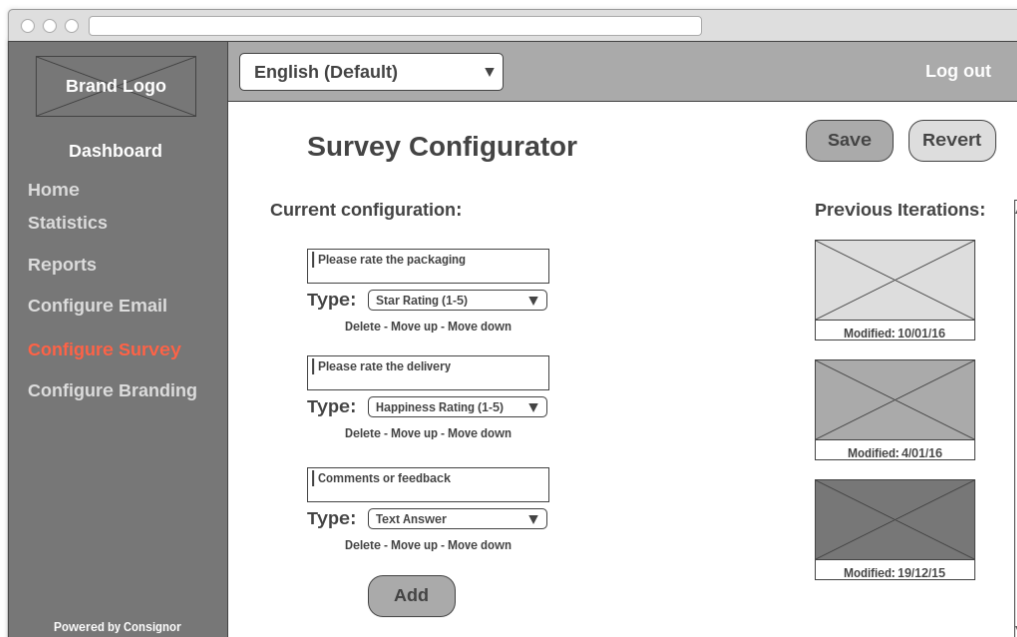


Figure 3: Wireframe for the survey configurator

We would also need to create a frontend for the surveys and tie it into the email engine. We created a wirefram proposal for the frontend:

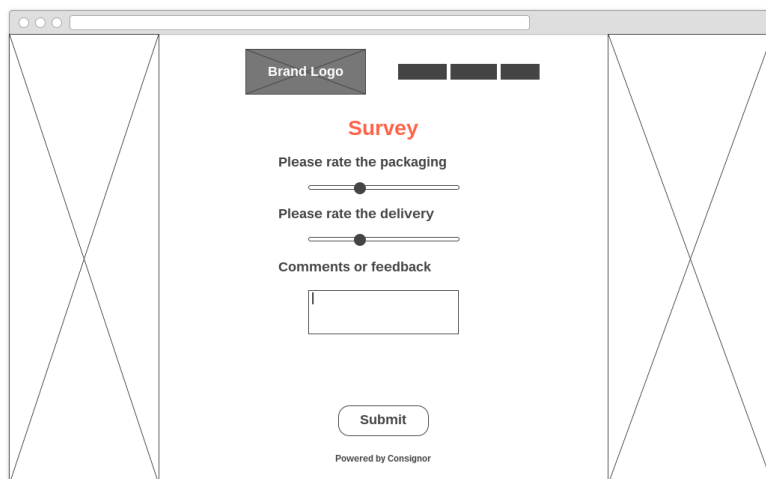


Figure 4: Wireframe for the survey front end

#### 2.3.4 Dashboard

The system dashboard is meant to give users insight into statistics surrounding email usage, survey feedback, etc.

### 3 Project Organization

#### 3.1 Responsibility

During development all group members are responsible for their own work, including reporting any difficulties they are facing to the group during the daily scrum, in addition to keeping occupied if the sprint backlog empties before the end of a sprint.

The following parties are involved in this project:

**Product Owner :**

Consignor Group AS, Bjrn E Pedersen

**Mentor :**

Frode Haug, NTNU

**Group Leader :**

Tobias Lmnerd Madsen

**CTO :**

Thomas Sem-Jacobsen

### **3.2 Routines & Rules**

See appendix A for group rules.

As for routines, we will stick to the following set of practices:

- All commits to repository should contain working and tested code only
- All code should be commented
- API Endpoints should be documented
- All team members meet at 10 am on weekdays
- As group leader and scrum master Tobias will be organizing all meetings with Consignor
- All team members are to log time spent on the project at the end of the day
- Thomas is responsible for booking a room for the team to work in
- Thomas is responsible for merging the work log of all team members

## **4 Planning, Monitoring & Reporting**

### **4.1 Timeline & Development Model**

When looking at what development methodology to follow, it was quickly revealed that following an agile approach was more suitable than a plan driven one. This is due to time constrictions, the need for a working prototype at the end of the semester and most importantly the agile willingness, and ability, to accommodate for changing requirements. This is important as even though the core functionality of our project is set, uncertainty lies in how these are developed, in addition to requirements beyond Consignor's initial vision.

There were two agile models that seemed especially attractive, Scrum and Kanban. On the surface the two look very similar, but in reality they have very differing philosophies.

Kanban offers a lot more freedom of choice during development than Scrum. Like Scrum, Kanban also utilize a product backlog, it does however not use a sprint backlog. Instead Kanban use a to do list, and unlike the sprint backlog this list is not static during development. Kanban offers users the ability to restructure the to do list by moving items between it and the backlog. This is because Kanban is not timeboxed, but instead a gradual process. Even though

this is a development principle the team would like to follow, ultimately we believe that our inexperience with Kanban could be detrimental to the project.

Scrum on the other hand is a lot more structured, and enforces certain rules and time constraints. Once a sprint is planned both the workload and the sprint duration is unchangeable. This allows the team to focus on the task at hand, instead of worrying about the larger picture. In addition, preventing the product owner from presenting new requirements during development, and disrupting the workflow of the team.

In the end scrum was chosen due to its greater focus on a structured workflow, as well as Consignor's previous experience with scrum. By following the scrum methodology every sprint will result in a working increment of the final product which will be used during meetings with both supervisor and Consignor to provide a status report. This will benefit the team greatly, as it gives a more accurate representation of how the team is performing.

For our configuration of scrum Bjrn E Pedersen and ystein Ranvik, representatives from Consignor, will take on the role of product owners. The Scrum Master will take on some of the responsibilities of the product owner, such as managing the product backlogs. This due to our desire to be self organizing, and not rely too heavily on Consignor. The role of Scrum Master was given to Tobias, as he is the only one in the group with experience using scrum in a work setting. Tobias also has previous work experience with Consignor, and such as Scrum Master facilitates communication between the scrum team and product owner.

When development starts we will be following a 2 week sprint cycle, but as we get further into development the Scrum Master has the authority to lengthen or shorten the sprint duration to suit the needs of the team. Due to time constraints and the likelihood of failed estimations, we will allow the team to continue working from the product backlog if the sprint backlog is emptied prior to the end of the current sprint. Which elements will be included in this way will be discussed by the team.

## 4.2 Meeting Schedule

Daily scrum meetings are held at 10 am on every weekday, except for Fridays when it's held at 12 pm due to additional classes, at NTNU and lasts 15 min. During these meetings we will be using the standup principle as this is to be a short meeting, and exists to inform the other members of their current status, and any obstructions they are currently facing.

Meetings with Consignor marks the end of the current sprint, and will be done mainly through Skype. These meetings will combine both the sprint planning meeting and sprint review meeting.

Every week on Tuesdays at 13:30 the group will be meeting with the supervisor to review the project status, and make sure the group is working at a steady pace so as to not fall behind. As we get further into development,

meetings with the supervisor may be less frequent as the group becomes more self organizing, and the need for weekly status reports is reduced.

## **5 Quality Assurance**

### **5.1 Documentation, Standardization & Source Code**

All C# code will be documented using Visual studios own documentation standards. This is done by using XML, and XML like tags to generate full source code documentation in XML files. In addition to being easy to use, the generated documentation is very easy to understand, due to the hierarchical structure of XML.

To document the endpoints of the Web API we will be using ApiExplorer, a ASP.NET Web API library which automatically generates help pages. These pages include detailed information on controllers with example requests and responses.

Further, the backend code in C# will follow Microsofts own C# coding conventions, while the frontend will follow Google's Style guide for HTML, CSS and JavaScript.

### **5.2 Tools & Strategy**

#### **5.2.1 Microsoft Visual Studio 2015**

Visual Studio is developed by Microsoft, and is the industry standard for developing windows based applications, and web development with the ASP.net framework. Visual studio will be the IDE used by all team members during development.

#### **5.2.2 Git and GitHub**

Git is a version control system which manages and stores revisions of projects in an online repository. GitHub is an online repository meant to be used in conjunction with Git. During the project Git and GitHub will be used to store any documents related to the bachelor project such as code and the final report.

#### **5.2.3 Teamwork.com**

Teamwork.com is a project management tool, best suited for small teams. It does lack some of the enterprise features of JIRA, but does everything else easier and smoother. We are using its 'milestone' feature to represent the sprints, and have integration with GitHub to create GitHub issues for each task due in Teamwork.com.

#### **5.2.4 Bower and NuGet**

Bower and NuGet are package managers for web development and the Microsoft development platform respectively.

### 5.2.5 LaTeX

LaTeX is designed to write technical and scientific documents, and uses syntax based language to specify the layout of the final document. LaTeX will be used during the course of this project to write any official documents relating to the final report.

### 5.2.6 Draw.io

Free, lightweight and easy to use online diagram software for making flow charts, process diagrams, UML, ER and network diagrams.

## 5.3 Risk Analysis

#	Issue	Probability	Impact
1	A group member gets sick	Medium	Medium
2	The project is not completed in time	Medium	Low
3	The sprint backlog is empty before the sprint is completed	Medium	Low
4	Uncompleted sprint elements at the end of a sprint	Medium	High
5	Teamwork.com server is down	Low	Medium
6	GitHub server is down	Low	Medium
7	Loss of uncommitted work	Low	High

#### Issue 1:

Sickness is bound to happen, and there is little we can do to prevent this. Instead, the use of a well defined sprint backlog means any sick members is aware of both their own work, as well as the work of other team members. Further, the daily scrum meeting will be held through Skype as to include all members.

#### Issue 2:

The project will be used as a proof of concept instead of a final solution. Due to our use of scrum, every sprint should conclude in a working increment of the final product, thus resulting in a working prototype by the deadline.

#### Issue 3:

Due to both our inexperience working on larger projects, and the technologies we will be using, wrong estimations are bound to happen. To reduce the margin of error, estimations will be done by the whole group and reviewed by product owner. If the issue is still present, we will allow the team member to add an item from the product backlog, as time constraints do not allow downtime.

#### Issue 4:

The same prevention technique will be done as for issue 4. If this happens, the remaining items will be returned to the product backlog. During the sprint review meeting the previous sprint will be discussed, with a focus on which items



took longer than estimated.

Issue 7:

We will be using the development practice of continuous integration with the use of Jenkins. This will ensure at most only a days work will be lost, in addition to removing a lot of overhead such as compiling, deployment and running tests.

## 6 Schedule

### 6.1 Gantt Scheme

The project schedule is currently planned as the following:

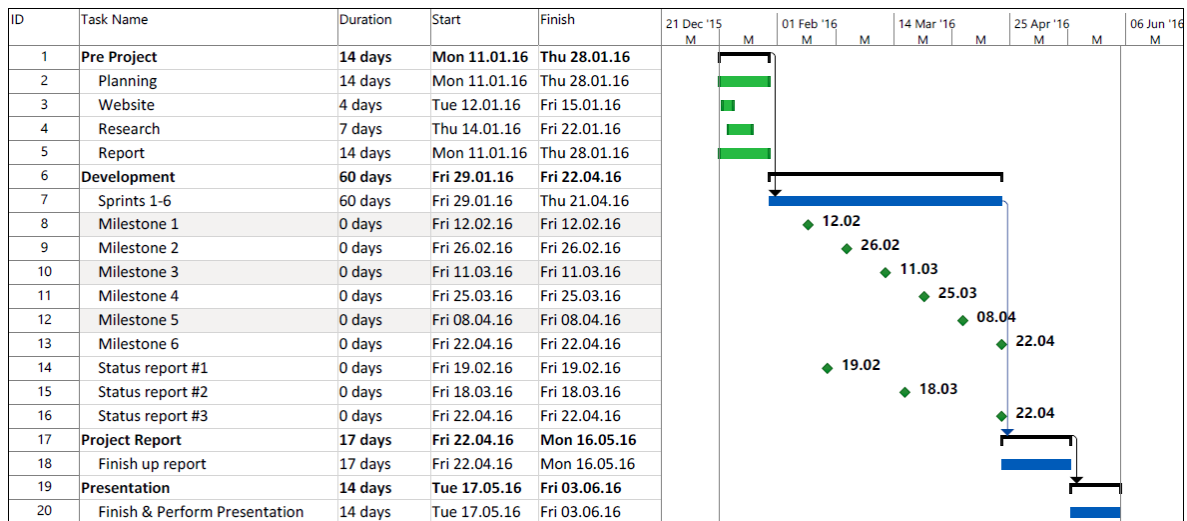


Figure 5: Gantt scheme with the projects phases

The development phase is in reality split into Scrum sprints, where each sprint lasts 2 weeks. This allows for a total of 6 sprints, with regular milestones. Development may start earlier than planned if the project plan gets finished and approved ahead of schedule.

### 6.2 Milestones

Since we have 6 Scrum sprints, it's also natural to define 6 major milestones as our goals. The milestones should represent major progress with the system.

1. Basic system structure, API routing and initial GUI
2. Database structure, complete GUI, basic email editor

3. API complete, email editor complete, branding complete
4. Email engine complete, survey editor and frontend complete
5. Statistics and report functionality
6. System testing complete, Product complete



## Appendix



## Meeting Summaries

Any meeting we had with Consignor in Oslo has been included in this appendix. Sprint meetings have not been included.

# Meeting Summary - 14/01-2016

*Consignor MGiNX - Bsc. Project 2016*

## Attending:

- Bjørn E Pedersen
- Tobias Lønnerød Madsen
- Øystein Ranvik
- Thomas Sem-Jacobsen

**Duration:** *3.5 hours*

## Summary

### Introduction

Got an introduction to the Consignor system, with an overview of the whole family of products.

### Project

Discussed what Consignor had in mind, then what we had in mind. Got the requirements and priorities confirmed. Also got the project agreement signed by both parties.

### Shipment status

We had envisioned receivers being sent to a shipment status page in our system, but this is not required, as the Portal product have this functionality. Idea scrapped for now.

### Email Editor & Engine

Added requirements for language support, variables based on actor configuration, DNS validation and more. Also clarified concepts and vocabulary surrounding the systems.

### Survey Configuration & Implementation

Needs a number of choices for response types, in a way that is intuitive for the sender to add to a survey. The surveys should have some permanent standard questions defined by us to measure general satisfaction etc. As with email components, survey components also needs to be able to adjust to local language. Bjørn and Øystein mentioned that they may be able to get some ideas and requirements from existing customers for this solution.

### Integration with Consignor

For this Bsc. project, we are to develop an independent module, without ties to Consignor. We will however receive sample data and examples of configurations to make our system as compliant as possible, for an easier integration at a later point.

### Architecture, Tools & Methods

As initially planned, we have settled on using Scrum with 2 week sprints. Regular sprint status meetings will be held with Bjørn and Øystein. Schedule is yet to be set. Git and Github for collaboration, and Visual Studio as our editor. The architecture has a frontend made with standard web technologies, and no javascript frameworks except for jQuery. The backend will be using ASP.net WebAPI as the framework and MariaDB with Entity Framework as datastoring.

### **Testing & Quality Assurance**

As initially planned, Bjørn also requested us to implement unit testing across the whole system. We have added this to the requirements.

### **Design**

Since this module likely will be integrated with the rest of the Consignor system, the design and aesthetics should be coherent with pre-existing design patterns used in the Consignor system.

### **Naming**

Consignor gave us the freedom to name it. We are currently naming it MGiNX (Mail Engine X).

# Meeting Summary - 07/04-2016

*Consignor MGiNX - Bsc. Project 2016*

## Attending:

- Bjørn E Pedersen
- Øystein Ranvik
- Lars Erik Fjørtoft
- Tellef Ormstad
- Peter T. Thomsen
- Tobias Lønnerød Madsen
- Thomas Sem-Jacobsen

**Duration:** *1.5 hours*

## Summary

This meeting was used to demo the product for Consignor, and to gather feedback on usability and functionality for the final sprint.

## Demo

Got to demo all of the features of the system to date, and got the following feedback:

## Dashboard

Felt overwhelming, may need to reduce footprint of alerts.

## Reports

May need a way to highlight unsatisfied responses to let the actor analyze individual answers. May need a way to filter answers based on shipment indicators like geography etc.

## Email Editor

Still need to implement button and text row functionality. There was some bugs with the token/variable functionality we need to resolve.

## Survey Editor

When creating a new language, the questions from the default language should be copied over to let the user translate immediately.

## Settings

Our approach to facilitate sending email on behalf of the actor is depending on DNS entries for SPF and DKIM in the actors DNS. This might not be optimal for all actors and an alternative to the solution using the actors email relay instead of our own was suggested as a solution. A intermediate solution will be implemented in the last sprint.

## Branding

Seems to work OK.



## Appendix

# ❧ H ❧

## Status Reports

(In Norwegian) Three status reports detailing the progress along the project timeline.

# Statusrapport nr. 1

## 1. Plannlegging

Fremdriftsplanen og forprosjektsplanen er ferdig en uke før frist, og dermed har utviklingen startet en uke før planlagt oppstart. I tillegg er kravspesifikasjonen er fullført og godtatt av oppdragsgiver.

## 2. Klargjøring av problemstilling

Ved utforming av kravspesifikasjon har vi fått god kontroll på oppdragets mål.

## 3. Løsningsmetode

Utviklingsmiljø er satt opp både for utviklere og på server-side. Server er testbar og online. Teknologistacken er bestemt og fungerende, og virker som vil la oss oppfylle alle satte mål.

## 4. Totalstatus

Vi har fått en god start på utviklingen, og ligger en uke før skjema med bi-ukentlige møter med oppdragsgiver.

## 5. Muligheter/Trusler/Problemer

Vi tar i bruk ASP.NET 5 som nylig ga ut release candidate, og det er av den grunn manglende dokumentasjon. Dette er en kilde til utfordringer når vi kommer dypere inn i backenden.

## 6. Hva er avsluttet

Vi er per i dag ferdig med andre sprint. Første sprint fullførte vi grunnleggende design og struktur på web applikasjonen, og test server har blitt satt opp. Andre sprint fikk vi satt opp mye av kommunikasjonen mellom frontend og backend, med mulighet for å sette ting som branding på emails og surveys, samt en tidlig versjon av email og surveyeditor.

## 7. Hva er under arbeid

Sprint 3 vil i hovedsak bestå av å koble resten av funksjonaliteten av frontend og backend sammen, samt resterende funksjonalitet i emaileditor. I forhold til milepælene satt for prosjektplanen ligger vi akkurat i rute.

## 8. Motivasjon

STRÅLENDE!!!



## 9. Veilederkontakt

Spesielt hjelpsomt i forprosjektsplanen.

# Statusrapport nr. 2

## 1. Fremdriftsplan

Sprint 5 begynner 21 mars, og avsluttes med en live demo for Consignor 7 april. Innen denne datoen bør all funksjonalitet være avsluttet slik at siste sprint kan brukes til generell oppussing og testing av systemet, i tillegg til endringer som Consignor ønsker.

## 2. Rapportskrivning

Grunnleggende struktur for rapporten er satt opp, og rapportskrivning begynner med sprint 5. Sprint 6 avsluttes 22 april, som markerer slutten for utviklingsprosessen, hvor rapportskrivning vil være hovedfokuset frem til 18 mai som er endelig innleveringsdato.

## 3. Totalstatus

Veldig mye av planlagt funksjonalitet er fullført, men mye er fremdeles løst tilkoblet og krever bedre intergrasjon. Etter planen ligger vi godt ann, utenom systemet for utsending av eposter som skal forsikre at eposter åptrer som troverdige. Her har det oppstått en del komplikasjoner grunnet bugs i RC1 av .Net Core, som har ført til at dette har tatt betydelig lenger tid en planlagt

## 4. Muligheter/Trusler/Problemer

Litt mangler og buggs med RC1 av .Net Core. Vi har laget løsninger for det midlertidig, men mye av disse problemene er fikset med RC2. RC2 var planlagt for midten av februar, men har blitt utsatt til ukjent tid.

## 5. Hva er avsluttet

Grunnleggende epost håndtering, språk støtte, logging av tilbakemeldinger og statistikk for tilbakemeldinger.

## 6. Hva er under arbeid

Sprint 5 har nylig begynt, og brukes hovedsakelig til å fullføre troverdig epost håndtering, samt survey og epost editor funksjonalitet, og lett rapportskrivning.

## 7. Motivasjon

Motivasjonen er fremdeles på topp, men vi merker at tiden begynner å bli knapp, og med dette øker stressnivået.

# Statusrapport nr. 3

## 1. Totalstatus

MGiNX systemet er nå fullført litt etter skjema, og vi rakk ikke like mye systemtesting som vi ville. Sprint 6 er nå avsluttet, og dermed også utviklingsprosessen. All planlagt funksjonalitet har blitt implementert utenom mulighet til å legge til egen avsender epost grunnet mangler i RC 1, men alt rundt er satt opp for å støtte det ved utgivelse av RC 2.

## 2. Fremdriftsplan

Fra den 25 april til innleveringsfristen den 18 mai vil all tid brukes til rapportskriving.

## 3. Rapportskriving

Endelig rapportstruktur er satt opp og sendes til veileder for kontroll. Den 25 april starter rapportskrivingen for fullt, og rapportens arbeidsoppgaver har blitt inndelt og tildelt med ukentlige tidsfrister til gruppemedlemene slik at alle vet hva de skal jobbe med.

## 4. Motivasjon

Som alltid er motivasjonen til gruppen på topp, og med kun 3 uker igjen kan vi endelig begynne å skimte lyset i enden av tunnelen.



## Appendix

# I

## Email Reader Compatability Tests

As part of testing and QA for the project, we tested our email templates in a multitude of email readers. The results are listed in this appendix, with a general overview first, and then screenshots of each email reader with the template as displayed.

Email Reader	Transmission	Formatting	Notes
Microsoft Outlook 2013	OK	OK	Padding on some elements missing (button)
Microsoft Outlook 2003	OK	OK	Padding on some elements missing (button), background color on some elements missing (separator)
Google Gmail	OK	OK	
Yahoo Mail	OK	OK (fixed)	Aters CSS attribute 'height' to 'min-height', fixed
Mozilla Thunderbird	OK	OK	

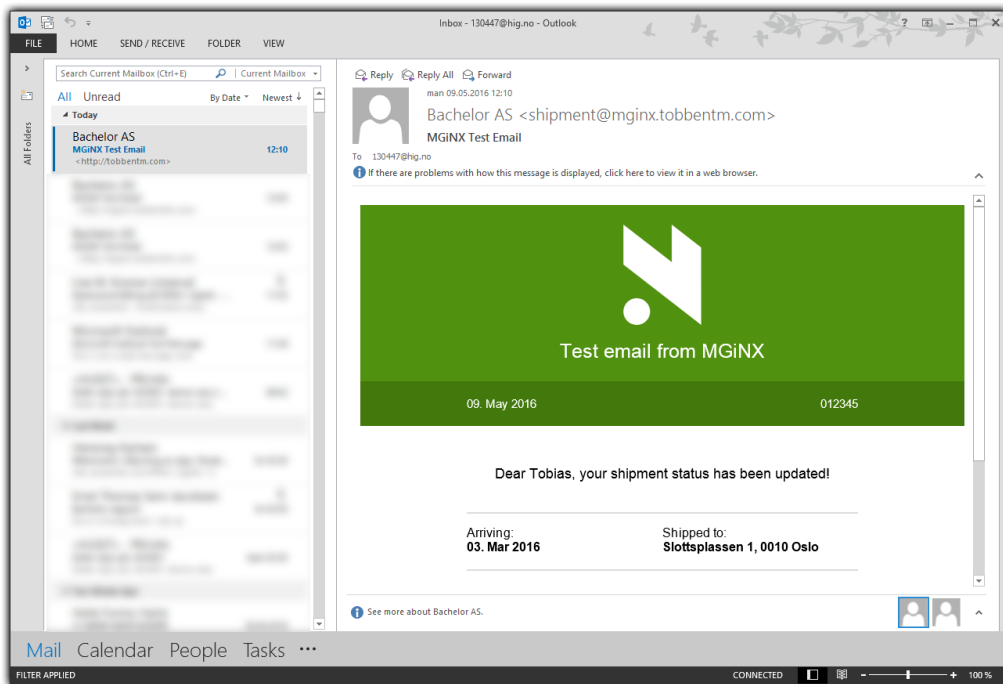


Figure 27: Screenshot of Microsoft Outlook 2013 displaying email.

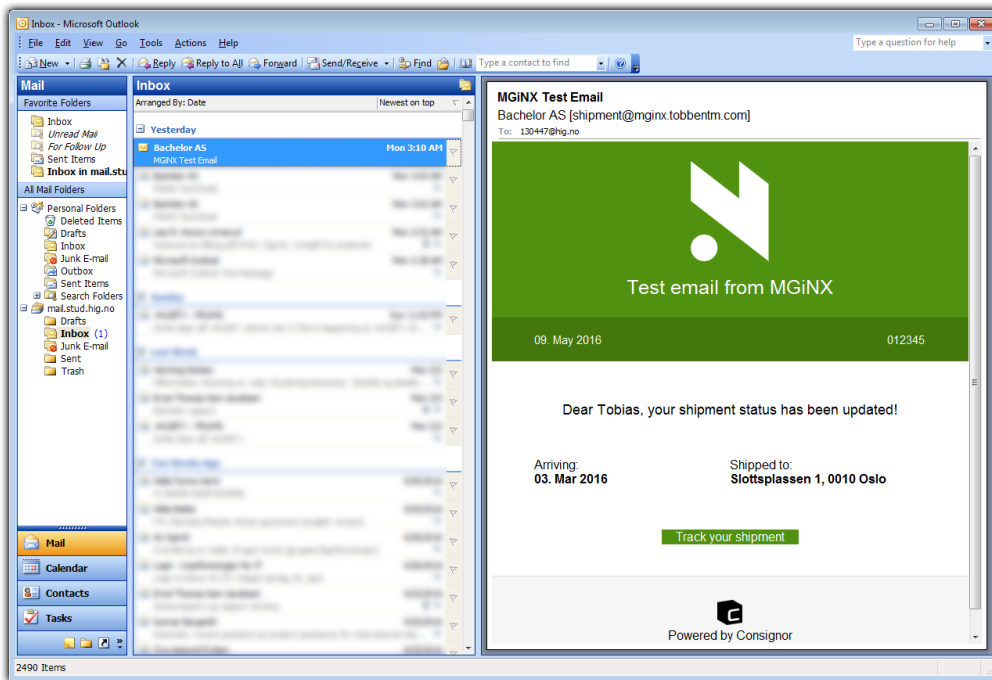


Figure 28: Screenshot of Microsoft Outlook 2003 displaying email.

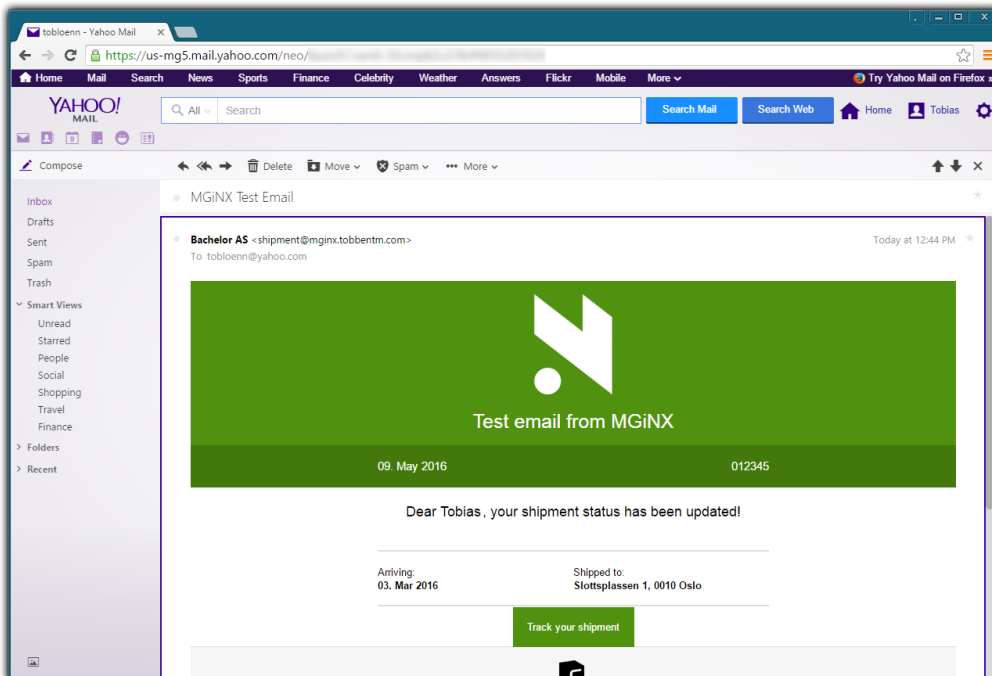


Figure 29: Screenshot of Yahoo Mail displaying email.

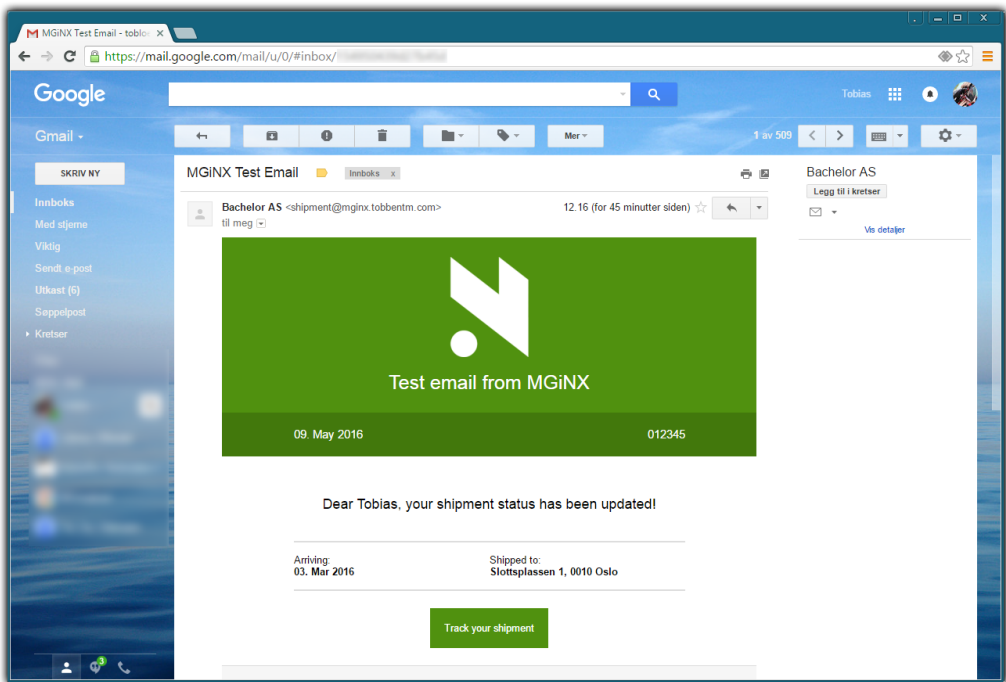


Figure 30: Screenshot of Google Gmail displaying email.

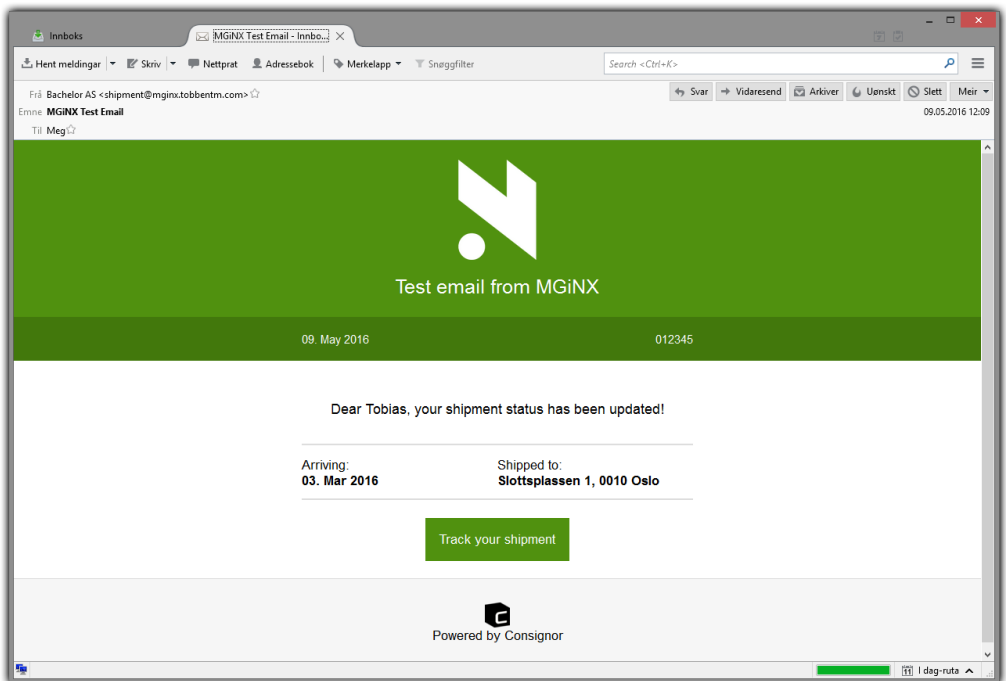


Figure 31: Screenshot of Mozilla Thunderbird displaying email.



## Appendix



## NuGet Configuration

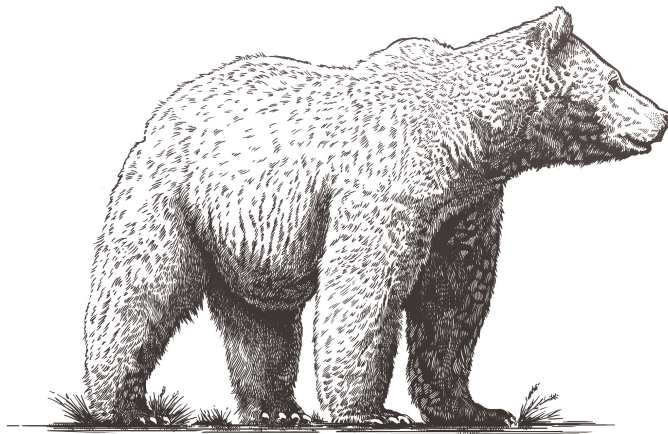
The following is the entirety of our project.json, showing project configuration and listing all libraries we use.



```
1 {
2   "version": "1.0.0-*",
3
4   "webroot": "wwwroot",
5
6   "compilationOptions": {
7     "emitEntryPoint": true
8   },
9
10  "dependencies": {
11    "Microsoft.ApplicationInsights.AspNet": "1.0.0-rc1",
12    "Microsoft.AspNet.IISPlatformHandler":
13      ↪ "1.0.0-rc1-final",
14    "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
15    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
16    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final",
17    "Microsoft.AspNet.Mvc.WebApiCompatShim":
18      ↪ "6.0.0-rc1-final",
19    "Microsoft.Extensions.Configuration.FileProviderExtensions":
20      ↪ "1.0.0-rc1-final",
21    "Microsoft.Extensions.Configuration.Json":
22      ↪ "1.0.0-rc1-final",
23    "Microsoft.Extensions.Logging": "1.0.0-rc1-final",
24    "Microsoft.Extensions.Logging.Console":
25      ↪ "1.0.0-rc1-final",
26    "Microsoft.Extensions.Logging.Debug": "1.0.0-rc1-final",
27    "Microsoft.AspNet.Diagnostics.Entity":
28      ↪ "7.0.0-rc1-final",
29    "Microsoft.AspNet.Identity.EntityFramework":
30      ↪ "3.0.0-rc1-final",
31    "Microsoft.Extensions.Configuration.Abstractions":
32      ↪ "1.0.0-rc1-final",
33    "Microsoft.Extensions.Configuration.EnvironmentVariables":
34      ↪ "1.0.0-rc1-final",
35    "EntityFramework.Commands": "7.0.0-rc1-final",
36    "EntityFramework7.Npgsql": "3.1.0-rc1-3",
37    "runtime.unix.System.Net.Security": "4.0.0-beta-23516",
38    "runtime.linux.System.Net.NetworkInformation":
39      ↪ "4.1.0-beta-*",
40    "xunit": "2.1.0",
41    "xunit.runner.dnx": "2.1.0-rc1-build204",
42    "LightMock.vNext": "1.0.1",
43    "MimeKit": "1.3.0-beta7",
44    "MailKit": "1.3.0-beta7",
45    "Portable.BouncyCastle": "1.8.1",
46    "Serilog.Framework.Logging": "1.0.0-rc1-final-10078",
47    "AngleSharp": "0.9.5"
48  },
49
50  "commands": {
51    "web": "Microsoft.AspNet.Server.Kestrel --server.urls
52      ↪ http://*:5000",
53    "ef": "EntityFramework.Commands",
```

```
43     "test": "xunit.runner.dnx"
44   },
45
46   "frameworks": {
47     "dnxc50": {
48       "dependencies": {
49         "PreMailer.Net.Core": "1.0.0-*"
50       }
51     }
52   },
53
54   "exclude": [
55     "wwwroot",
56     "node_modules"
57   ],
58   "publishExclude": [
59     "**.user",
60     "**.vspscc"
61   ]
62 }
```

---



## Appendix



## Bower Configuration

The following is the entirety of our Bower configuration, listing all libraries we use.

```
1 {
2   "name": "MGiNX",
3   "private": true,
4   "dependencies": {
5     "bootstrap": "~3.3.6",
6     "startbootstrap-sb-admin-2": "~1.0.8",
7     "mjolnic-bootstrap-colorpicker": "~2.3.0",
8     "Sortable": "~1.4.2",
9     "html2canvas": "~0.4.1",
10    "es6-promise": "~3.0.2",
11    "bootstrap-star-rating": "~3.5.7",
12    "js-cookie": "~2.1.0",
13    "Medium.js": "medium.js#~1.0.1",
14    "bootstrap-validator": "0.9.0",
15    "chosen": "^1.5.1",
16    "jquery": "2.1.4",
17    "opentip": "^2.4.6"
18  },
19  "resolutions": {
20    "jquery": "2.1.4",
21    "font-awesome": "~4.2"
22  }
23 }
```

---



## Appendix



### UserController.cs

The following is an example of the controllers we use in the application, in this case serving user objects.

---

```

using System.Collections.Generic;
using System.Net;

using Microsoft.AspNet.Mvc;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json.Linq;

using MGiNX.API.User.Models;
using MGiNX.API.Authentication;

namespace MGiNX.API.User.Controllers
{
    [Route("api/[controller]")]
    public class UserController : Controller
    {
        /// <summary>
        /// IEmailLayoutRepository is an interface used to
        /// ↪ separate database operations from controller and
        /// ↪ model logic.
        /// </summary>
        /// <seealso cref="IUserRepository"/>
        public IUserRepository _repository;
        private readonly ILogger _logger;

        /// <summary>
        /// Dependency Injection is used to retrieve resources
        /// ↪ the controller will utilize.
        /// </summary>
        /// <seealso cref="Startup"/>
        public UserController(IUserRepository repository,
            ↪ ILogger<UserController> logger)
        {
            _repository = repository;
            _logger = logger;
        }

        /// <summary>
        /// Retrieves the UserItem with property Id equal to
        /// ↪ UserId.
        /// </summary>
        /// <returns>
        /// HttpUnauthorized if the cookie is invalid.
        /// HttpNotFound if the UserItem does not exist.
        /// UserItem if the UserItem exist.</returns>
        [HttpGet]
        public IActionResult GetUser()
        {
            int userId;
            try
            {
                userId =
                    ↪ CookieAuthentication.Verify(HttpContext.Request.Cookies);
            }
        }
    }
}

```

```

    }
    catch (CookieAuthenticationFailedException)
    {
        Response.StatusCode =
            ↪ (int)HttpStatusCode.Unauthorized;
        return HttpUnauthorized();
    }

    UserItem user = _repository.FindUser(userId);
    if (user == null)
    {
        Response.StatusCode =
            ↪ (int)HttpStatusCode.NotFound;
        return HttpNotFound();
    }

    return new ObjectResult(user);
}

/// <summary>
/// Takes a Json Object containing a string of
    ↪ languages. These languages are then added
/// to the current users SettingItem.
/// </summary>
/// <param name="payload">Json Object containing user
    ↪ data</param>
/// <returns>
/// HttpNotFound if the cookie is invalid.
/// HttpNotFound if the UserItem does not exist.
/// SettingsItem if the Language was successfully
    ↪ added.</returns>
[HttpPost("add/language")]
public IActionResult AddLanguage([FromBody] JObject
    ↪ payload)
{
    int userId;
    try
    {
        userId =
            ↪ CookieAuthentication.Verify(HttpContext.Request.Cookies);
    }
    catch (CookieAuthenticationFailedException)
    {
        Response.StatusCode =
            ↪ (int)HttpStatusCode.Unauthorized;
        return HttpUnauthorized();
    }

    UserItem user = _repository.FindUser(userId);
    if (user == null)
    {
        Response.StatusCode =
            ↪ (int)HttpStatusCode.NotFound;
    }
}

```

```
        return HttpNotFound();
    }

    string iso = payload.Value<string>("lang");

    List<string> langs = new
        ↪ List<string>(user.Settings.Languages);
    langs.Add(iso);
    user.Settings.Languages = langs.ToArray();

    _repository.SaveChanges();

    return new ObjectResult(user.Settings);
}
}
```

---





## Appendix



**M**



**UserRepository.cs**

The following is an example of the repositories we use in the application, in this case serving user objects.

```

using System;
using System.Linq;

using Microsoft.Data.Entity;

using MGiNX.API.Branding.Models;
using MGiNX.API.Setting.Models;
using MGiNX.API.Utils;
using MGiNX.DatabaseContext;

namespace MGiNX.API.User.Models
{
    public class UserRepository : IUserRepository
    {
        private readonly PostgreSQLContext _dbContext;

        public UserRepository(PostgreSQLContext dbContext)
        {
            _dbContext = dbContext;
        }

        /// <summary>
        /// Attempts to add a new UserItem to the database with
        /// ↪ property Id equal to the userId parameter.
        /// If the item is not added the changes are reverted.
        /// </summary>
        /// <param name="userId"></param>
        /// <returns>
        /// Null if there was an error adding the new user.
        /// UserItem if the user was successfully
        /// ↪ added.</returns>
        private UserItem AddUser(int userId)
        {
            //TODO: Ensure creation of new
            ↪ setting/branding/survey stuff
            UserItem user = new UserItem { Id = userId, Events
            ↪ = Constants.Events };
            user.Branding = new BrandingItem { };
            user.Settings = new SettingItem { DefaultLanguage =
            ↪ "en", Languages = new string[] { "en" } };
            try
            {
                _dbContext.BrandingItems.Add(user.Branding);
                _dbContext.SettingItems.Add(user.Settings);
                _dbContext.Users.Add(user);
                _dbContext.SaveChanges();
            } catch (InvalidOperationException ex)
            {
                System.Diagnostics.Debug.WriteLine("Exception
                ↪ occurred during user creation: " +
                ↪ ex.Message);
                return null;
            } catch (DbUpdateException ex)

```

```

    {
        System.Diagnostics.Debug.WriteLine("Exception
            ↪ occurred during user creation: " +
            ↪ ex.Message + ", reverting changes...");
        _dbContext.BrandingItems.Remove(user.Branding);
        _dbContext.SettingItems.Remove(user.Settings);
        _dbContext.Users.Remove(user);
        _dbContext.SaveChanges();
        return null;
    }
    return user;
}

/// <summary>
/// Attempts to find the UserItem with property Id
    ↪ equal to the userId parameter. If the User does
    ↪ not exist
/// add the user to the database as a new user.
/// </summary>
/// <param name="userId">Id equal to UserItem property
    ↪ Id</param>
/// <returns>
/// UserItem if the item exists, or was added.
/// Null if the item does not exist.</returns>
public UserItem FindUser(int userId)
{
    UserItem user;
    try
    {
        user = _dbContext.Users
            .Include(u => u.Branding)
            .Include(u => u.Settings)
            .First(u => u.Id == userId);
    }
    catch (InvalidOperationException ex)
    {
        System.Diagnostics.Debug.WriteLine("Exception
            ↪ occurred during user retrieval: " +
            ↪ ex.Message);
        user = null;
    }
    if(user == null)
    {
        user = AddUser(userId);
    }
    return user;
}

/// <summary>
/// Implements the changes made to the database.
/// </summary>
public void SaveChanges()
{

```

```
        _dbContext.SaveChanges();  
    }  
}  
}
```

---



## Appendix



## Sprint Log

## Sprint 1 — Task List Report

## Completed Tasks

Task	Start Date	Date Due	Responsible	Assigned By	Priority	Progress	Status
Set up test server		05 Feb (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 25 Jan (2016) by Tobias Lønnerød M.
[Web] GUI for Dashboard Webpage for dashboard, should serve as a foundation for all other pages		05 Feb (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 27 Jan (2016) by Tobias Lønnerød M.
[Web] GUI for Email Editor		05 Feb (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 04 Feb (2016) by Tobias Lønnerød M.
[Web] GUI for Survey Editor		05 Feb (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 01 Feb (2016) by Tobias Lønnerød M.
[Web] GUI for Reports		05 Feb (2016)	Tobias Lønnerød M.	Ernst Thomas S.		100%	Completed 08 Feb (2016) by Tobias Lønnerød M.
[Web] GUI for Branding		05 Feb (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 29 Jan (2016) by Tobias Lønnerød M.
[Web] GUI for Settings		05 Feb (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 27 Jan (2016) by Tobias Lønnerød M.
[Sys] Application Structure		05 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 26 Jan (2016) by Ernst Thomas S.
[API] API Routing		05 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 27 Jan (2016) by Ernst Thomas S.
[API] Testing Tools		05 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 08 Feb (2016) by Ernst Thomas S.
[Sys] Database Integration Using Entity Framework		05 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 01 Feb (2016) by Ernst Thomas S.

## Sprint 2 — Task List Report

## Completed Tasks

Task	Start Date	Date Due	Responsible	Assigned By	Priority	Progress	Status
[GUI] Polish and Adjust		19 Feb (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 16 Feb (2016) by Tobias Lønnerød M.
[API] Email Editor Endpoint		19 Feb (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 19 Feb (2016) by Tobias Lønnerød M.
[Editor] User can add predefined tokens to emails		19 Feb (2016)	Tobias Lønnerød M.	Ernst Thomas S.		100%	Completed 17 Feb (2016) by Tobias Lønnerød M.
[Editor] User can save and load email configurations		19 Feb (2016)	Tobias Lønnerød M.	Ernst Thomas S.		100%	Completed 19 Feb (2016) by Tobias Lønnerød M.
[Fronddend] API Hook-up Survey Editor		19 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 19 Feb (2016) by Ernst Thomas S.
[Fronddend] API Hook-up Branding		19 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 12 Feb (2016) by Ernst Thomas S.
[Fronddend] API Hook-up Settings		19 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 16 Feb (2016) by Ernst Thomas S.
[Editor] Create a set of predefined templates the user can use to specify layout of emails		19 Feb (2016)	Tobias Lønnerød M.	Ernst Thomas S.		100%	Completed 17 Feb (2016) by Tobias Lønnerød M.
[API] Survey Editor Endpoint		19 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 19 Feb (2016) by Ernst Thomas S.
[API] Settings Endpoint		19 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 16 Feb (2016) by Ernst Thomas S.
[API] Branding Endpoint		19 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 12 Feb (2016) by Ernst Thomas S.
[Sys] Database POCOs		19 Feb (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 12 Feb (2016) by Ernst Thomas S.

## Sprint 3 — Task List Report

## Completed Tasks

Task	Start Date	Date Due	Responsible	Assigned By	Priority	Progress	Status
[API] User Endpoint (Global Settings)		04 Mar (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 25 Feb (2016) by Tobias Lønnerød M.
[Refactor] Cookie Authentication		04 Mar (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 22 Feb (2016) by Tobias Lønnerød M.
[Refactor] GUI Dialogs		04 Mar (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 22 Feb (2016) by Tobias Lønnerød M.
[Editor] User can choose the language for a specific email configuration		04 Mar (2016)	Tobias Lønnerød M.	Ernst Thomas S.		100%	Completed 23 Feb (2016) by Tobias Lønnerød M.
[Email] Engine Mockup		04 Mar (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 04 Mar (2016) by Tobias Lønnerød M.
[API] Dashboard Endpoint		04 Mar (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 23 Feb (2016) by Ernst Thomas S.
[Fronddend] API Hook-up Dashboard		04 Mar (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 23 Feb (2016) by Ernst Thomas S.
[Survey] Load config for user		04 Mar (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 24 Feb (2016) by Ernst Thomas S.
[Survey] Generate a unique one time access url for the survey webpage		04 Mar (2016)	Ernst Thomas S.	Ernst Thomas S.		100%	Completed 24 Feb (2016) by Ernst Thomas S.
[Survey] Log survey answers		04 Mar (2016)	Ernst Thomas S.	Ernst Thomas S.		100%	Completed 01 Mar (2016) by Ernst Thomas S.



## Sprint 4 — Task List Report

## Completed Tasks

Task	Start Date	Date Due	Responsible	Assigned By	Priority	Progress	Status
[Fronddend] API Hook-up Reports		18 Mar (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 10 Mar (2016) by Ernst Thomas S.
[API] Survey Report Endpoint		18 Mar (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 10 Mar (2016) by Ernst Thomas S.
MTA System		18 Mar (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 18 Mar (2016) by Tobias Lønnerød M.
[Common] Adding language (dialog)		18 Mar (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 14 Mar (2016) by Tobias Lønnerød M.
[API] Implement Logging		18 Mar (2016)	Ernst Thomas S.	Ernst Thomas S.		100%	Completed 14 Mar (2016) by Ernst Thomas S.

Generated for Ernst Thomas Sem-Jacobsen at 15:43 14/05/2016

## Sprint 5 — Task List Report

## Completed Tasks

Task	Start Date	Date Due	Responsible	Assigned By	Priority	Progress	Status
[Language] Ability to delete language		08 Apr (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 05 Apr (2016) by Ernst Thomas S.
[Engine] Tracking/Survey URL Generation		08 Apr (2016)	Anybody	Tobias Lønnerød M.		100%	Completed 11 Apr (2016) by Tobias Lønnerød M.
[Editor] Copy-From (Copy content from another event/language)		08 Apr (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 06 Apr (2016) by Tobias Lønnerød M.
[Dashboard] Status messages		08 Apr (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 06 Apr (2016) by Ernst Thomas S.
[Unit Testing] Extend and automate tests		08 Apr (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 27 Mar (2016) by Ernst Thomas S.
[Survey] User can choose the language for a specific survey configuration		08 Apr (2016)	Ernst Thomas S.	Ernst Thomas S.		100%	Completed 29 Mar (2016) by Ernst Thomas S.
[Editor] User can add and remove attachments to emails		08 Apr (2016)	Tobias Lønnerød M.	Ernst Thomas S.		100%	Completed 05 Apr (2016) by Tobias Lønnerød M.
[Email] Finish MUA		08 Apr (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 01 Apr (2016) by Tobias Lønnerød M.
[Email/MTA] Fix SPF errors and verify SSL		08 Apr (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 22 Mar (2016) by Tobias Lønnerød M.

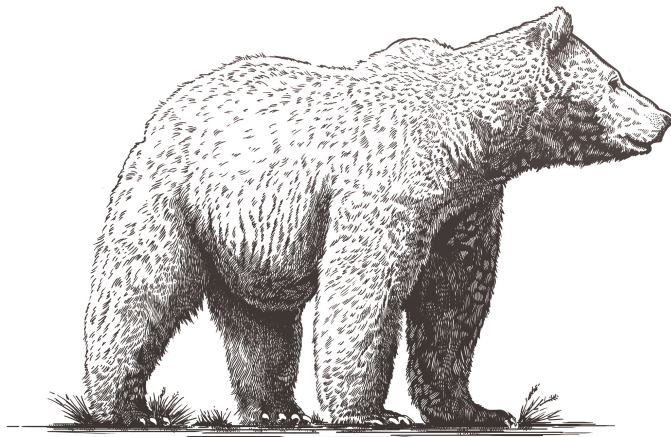
Generated for Ernst Thomas Sem-Jacobsen at 15:44 14/05/2016

## Sprint 6 — Task List Report

## Completed Tasks

Task	Start Date	Date Due	Responsible	Assigned By	Priority	Progress	Status
[Build] Automate build for docker and database update		22 Apr (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 25 Apr (2016) by Tobias Lønnerød M.
[Editor] Survey Button + Text/image rows/columns		22 Apr (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 21 Apr (2016) by Tobias Lønnerød M.
[Engine] Complete email engine		22 Apr (2016)	Tobias Lønnerød M.	Tobias Lønnerød M.		100%	Completed 25 Apr (2016) by Tobias Lønnerød M.
[Meta] Document undocumented code		22 Apr (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 12 Apr (2016) by Ernst Thomas S.
[Meta] Document API interface		22 Apr (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 15 Apr (2016) by Ernst Thomas S.
[Meta] Finish Unit Testing all modules		22 Apr (2016)	Ernst Thomas S.	Tobias Lønnerød M.		100%	Completed 12 Apr (2016) by Ernst Thomas S.

Generated for Ernst Thomas Sem-Jacobsen at 15:44 14/05/2016



## Appendix



## Swagger UI

# MGiNX

Email editor API

Find out more about Swagger

<http://swagger.io>

## branding : Everything about branding

Show/Hide | List Operations | Expand Operations

**GET** /branding Get branding by user

**POST** /branding Add or update a branding item on the server

## dashboard : Everything about dashboard

Show/Hide | List Operations | Expand Operations

**GET** /dashboard Gets the configuration status of a user

## emailLayout : Everything about email

Show/Hide | List Operations | Expand Operations

**GET** /emailLayout/{eventId}/{language} Gets the currently active emailLayout for the given user filtered by event and language

**GET** /emailLayout/{eventId}/{language}/config Gets the 6 most recent email layouts for the given user filtered by event and language

**POST** /emailLayout/{eventId}/{language}/add Creates a new email layout for the given user on event and language

**DELETE** /emailLayout/{eventId}/{language}/{date} Deletes the email layout and email configuration for the given user filtered by EventId and Language

**POST** /emailLayout/{eventId}/{language}/attachment Saves an attachment for the given user on event and language

**DELETE** /emailLayout/{eventId}/{language}/attachment/{index}/delete Deletes a given attachment

**GET** /emailLayout/{eventId}/{language}/attachment/{index}/view Gets a given attachment

## report : Everything about report

Show/Hide | List Operations | Expand Operations

**GET** /report Get report item for current user

**GET** /report/{guid} Get all text answers for a question

## setting : Everything about setting

Show/Hide | List Operations | Expand Operations

**GET** /setting Get branding by user

**POST** /setting Add or update setting item on the server

## survey : Everything about survey

Show/Hide | List Operations | Expand Operations

**GET** /survey/create/{language} Creates and returns a survey url

**GET** /survey/config/{language} Gets survey for user on language

**DELETE** /survey/{guid} Deletes a survey item for the given user on language

**GET** /survey/{guid} Get all questions associated with the survey as well as branding

**POST** /survey/{guid} Adds all answers to the server

**POST** /survey Add or update a survey item on the server

## user : Everything about user

Show/Hide | List Operations | Expand Operations

**GET** /user Get user item for active user

**POST** /user/add/language Add language to user



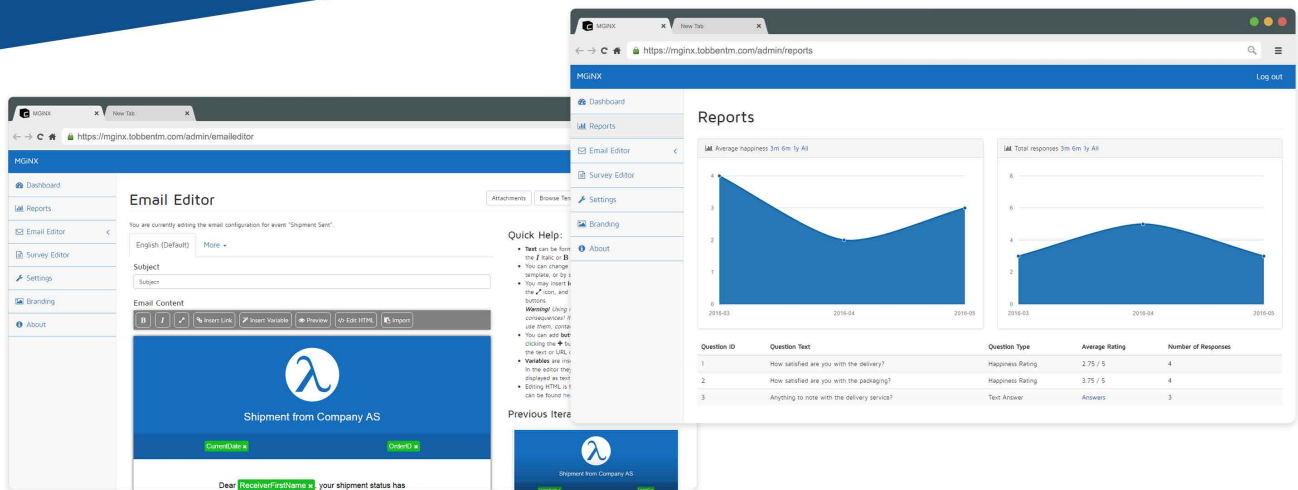
## Appendix



## Project Poster

# MGiNX

## Creating a modern platform for email delivery



Thesis describing the creation of a modern platform for managing trustworthy email delivery, email template editing and survey management

Tested compatability with all email clients, using industry grade email verification and encryption



Faculty of Computer Science and Media Technology

Using the latest web and backend technologies, including ASP.Net Core, Entity Framework 7, PostgreSQL, Docker and Postfix

### Meet the team



Tobias  
13HBIDATA



Thomas  
13HBIDATA

<http://tobbentm.com/bsc>