



NTNU – Trondheim
Norwegian University of
Science and Technology

Developing a Content Management System with a Microservice Architecture

Group 1

Aleksander Hansen

Haakon Johansen Jonsson

Pål Karlsrud

Ole Martin Knurvik

Nicklas Grimstad Nilsen

Håkon Larsen Solbjørg

Thomas Hansen (Dropped out 7th of March)

May 2016

Department of Computer and Information Science
Norwegian University of Science and Technology

Professor: Monica Divitini

Supervisor: Katja Abrahamsson

Client: Mohsen Anvaari, Trondheim Municipality

Abstract

A thesis for the Bachelor in Informatics study program at NTNU, with the goal of demonstrating the usage of microservices for Trondheim Municipality. The thesis was completed during spring 2016.

The client, Trondheim Municipality, is as of spring 2016 using multiple monolithic systems. In contrast, Oslo Municipality is actively expanding infrastructures into microservices (Gundelsby, 2014). The use of microservices aids in simplifying maintenance, accelerates development of new features and enables ad-hoc repair of software defects without bringing down entire systems.

The project purported to implement a demonstration of a Content Management System for Trondheim Municipality, using a microservice architecture. A monolithic implementation was ordered from an external company by Trondheim Municipality prior – the microservice implementation being an exploration into a potentially more advantageous solution.

Contents

Abstract	i
List of Figures	v
List of Tables	vii
Acronyms	ix
1 Introduction	1
1.1 Customer	1
1.2 The team	2
1.3 The project goal	2
2 Requirements	3
2.1 Selecting requirements	3
2.2 Functional Requirements	4
2.2.1 Content related services	4
2.2.2 Authentication	6
2.2.3 Search related services	7
2.2.4 Service-to-service communication	8
2.2.5 Status	8
2.3 Non-functional requirements	8
3 Project management	11
3.1 Risks	11
3.2 Absence	13
3.3 Time frame	14
3.4 Estimation	15
3.5 The team	16
3.6 Team organisation	16
3.6.1 Roles	17
3.6.2 Responsibilities	17
3.7 Software development process	17
3.8 Iterations	19

3.8.1	Iteration 1	19
3.8.2	Iteration 2	22
4	Alternative solutions	24
4.1	Technologies	24
4.1.1	Deployment alternatives	25
4.1.2	API structure	26
4.1.3	Service-to-service communication alternatives	26
4.2	Development method alternatives	29
5	Tools and technologies	31
5.1	Development tools	31
5.1.1	Git	31
5.1.2	Go	31
5.1.3	Python	31
5.1.4	JavaScript	32
5.1.5	MongoDB	33
5.1.6	PostgreSQL	33
5.2	Deployment tools	33
5.2.1	Docker	33
5.2.2	Docker Compose	34
5.2.3	Nginx	34
5.2.4	Teamcity	34
5.2.5	Jenkins	34
5.2.6	Travis CI	34
5.2.7	Logstash	34
5.3	Testing	35
5.4	Project management tools	35
5.5	Communication tools	36
6	Architecture	37
6.1	System architecture	37
6.2	How the services communicate	38
6.3	The architecture of each microservice	39
6.3.1	Front-end service	40
6.3.2	Publishing service	41
6.3.3	Template service	41
6.3.4	Status services	41
6.3.5	Authentication service	41
6.3.6	Index service	42

6.3.7	Spelling service	42
6.3.8	Search service	43
6.4	How the architecture scales	44
6.4.1	Stress testing	46
6.5	Deployment	47
6.6	How the system could be improved	47
6.6.1	Architecture	48
6.6.2	Deployment	48
6.6.3	Services	49
7	Testing	50
7.1	Test strategy	50
7.1.1	Project environment	51
7.1.2	Product elements	51
7.1.3	Quality criteria	51
7.1.4	Test techniques	51
7.2	Testing microservices	52
7.3	Automated tests	52
7.3.1	Continuous Integration	53
7.3.2	Continuous Deployment	53
7.3.3	Types of tests to automate	53
7.4	Testing of third party software	54
7.5	Testing non-functional requirements	54
7.6	Acceptance test	55
8	Summary and Conclusions	56
8.1	Architecture	56
8.1.1	Architectural advantages	56
8.1.2	Architectural disadvantages	56
8.2	Problems and solutions	57
8.2.1	Time schedule	57
8.2.2	Abroad	58
8.2.3	Group member leaving	58
8.3	Lessons learnt	58
8.3.1	Project organisation	58
8.3.2	Development	59
8.4	Conclusion	59
8.5	Summary	60

A	Requirements	xiv
A.1	Content module	xiv
A.2	Authentication module	xv
A.3	Search module	xv
B	Setup guide	xvi
C	User manual	xix
C.1	Creating an article	xix
C.2	Searching	xxiii
C.3	Viewing service status	xxvi

List of Figures

3.1	Planned absence	14
3.2	Activity plan with tasks and deadlines. Arrows indicates dependencies.	15
3.3	Development method	19
4.1	Example of communication using direct communication.	27
4.2	Illustration of communication using a message broker.	28
4.3	Communication between services using the DHT architecture.	29
6.1	Connections between different microservices and other components.	38
6.2	Flow of requests during a query.	39
6.3	Overview of the original design of the content services.	40
6.4	Overview of the refactored design of the content services.	40
6.5	The flow of OAuth2 is as described in the OAuth2 RFC (Hardt, 2012), section 1.2.	42
6.6	Sequence diagram of a complete search query. A list of results and a list of lists of spelling feedback (indexed by position in query sentence) is returned.	44
6.7	The scalability of the front-end service	45
6.8	The scalability of the publishing service	45
6.9	The performance of the front-end service under normal load	46
6.10	How the front-end service performs under stress	46
6.11	Deployment of services.	47
8.1	Example of our schedule. One row for each team member.	57
C.1	Showing location of the editor button in main menu	xx
C.2	View for step 2	xxi
C.3	View for step 2	xxi
C.4	Showing location of the template drop-down list	xxii
C.5	Showing location of the submit button	xxii
C.6	Showing location of the submit button	xxiii
C.7	Showing location of the search button in main menu	xxiv
C.8	Word completion suggestion	xxv

C.9 Search results	xxv
C.10 Full article	xxvi
C.11 Hidden status button	xxvii
C.12 Status overview. The symbol in front of the name represents its state. Check mark means available and an error symbol means it's unavailable.	xxvii
C.13 Service documentation	xxviii

List of Tables

2.1	User stories for the content services	6
2.2	User stories for the authentication service	7
2.3	User stories for the search, index, and spelling services	7
2.4	Functional requirements for the communication between services	8
2.5	Functional requirements for the status service	8
2.6	Non-functional requirements	9
2.7	Desired Non-functional qualities	10
3.1	Risk analysis	13
3.2	The team, knowledge mapping	16
3.3	User stories for iteration 1	21
3.4	User stories for iteration 2	23
4.1	Pros and cons of direct communication	26
4.2	Pros and cons of message brokers	27
4.3	Pros and cons of a fully distributed system	29
7.1	Code coverage statistics for the various services	54
A.1	Functional requirements for the content services	xiv
A.2	Functional requirements for the authentication service	xv
A.3	Functional requirements for the search service	xv

Acronyms

API Application programming interface. 8, 25–27, 37, 40, 42, 47, 48, 54, 59

CI Continuous Integration. 33, 34, 52

CMS Content Management System. 2, 36

CTO Chief Technology Officer. 17, 51

DevOps Development and Operations. 11, 32, 49, 51

DHT Distributed Hash Table. 8, 28, 29, 31, 37, 47

DRY Don't Repeat Yourself. 26

ECTS European Credit Transfer and Accumulation System. 1

HTML HyperText Markup Language. 40

HTTP HyperText Transfer Protocol. 32, 37, 40, 43

IEC International Electrotechnical Commission. 8, 9

IP Internet Protocol. 8, 21, 26–29, 37, 38, 47

ISO International Organization for Standardization. 8, 9

JSON JavaScript Object Notation. 26, 32, 36, 40

NTNU Norwegian University of Science and Technology. 1, 2

REST Representational state transfer. 8, 9, 26, 37, 47, 54

RFC Request For Comments. 41

RPS Requests Per Second. 44

SOA Service Oriented Architecture. 1, 48, 55

SOAP Simple Object Access Protocol. 26

TRK Trondheim kommune (Trondheim Municipality). 1–3, 54

UI User Interface. 6, 16, 17, 23

UX User eXperience. 17

VCS Version Control System. 51

VM Virtual Machine. 25

WIP Work In Progress. 18

WSDL Web Services Description Language. 26

WYSIWYG What You See Is What You Get. 5, 6, 21, 31, 39

XML eXtensible Markup Language. 26

Chapter 1

Introduction

This project is a part of IT2901 - Informatics Project II. The course aims to teach about process and organisation in software development projects. The course grants 15 credits (ECTS) and serves as the bachelor assignment for Informatics bachelor degrees at NTNU.

Students partaking in the course were divided into groups and assigned an actual customer, who provided a project to be developed and documented by the students – this report being the documentation of the development of a Content Management System using microservices made for the municipality of Trondheim.

The report was written for the faculty staff and the customer. It contains some terminology assumed known to the reader. Acronyms, abbreviations, and some terms used are listed in the acronyms section.

1.1 Customer

Trondheim Municipality, with around 13 000 employees, delivers various services to its employees and to the citizens of the municipality through construction, roads, education, health care, environment, politics, and cultural domains. An ever-growing set of these services have been and are in the process of being digitised. New services are intended to be web- and app based solutions, and to be highly integrated with each other in an attempt to reduce information redundancy and increase reusability.

As an architecture principle, all new systems in TRK should be developed based on the SOA-pattern. However, SOA is in the process of changing its paradigm from a monolithic structure to a microservice structure. The microservice concept has, as of 2016, been the subject of

numerous heavy discussions with advocates on both sides.

Microservices are however not only a theoretical concept. Oslo Municipality is one of the adopters of this concept, and has developed around 150 microservices (Gundelsby, 2014). Trondheim Municipality is interested in seeing how a microservice oriented system functions compared to a monolithic system.

TRKs new CMS ordered from an external customer will be a monolithic system. In this proof-of-concept project, there was expressed desire to see how a microservice CMS developed by IT2901 students would perform regarding time-to-market, performance, modularity, and scalability (Anvaari, 2016).

1.2 The team

The team consists of six students at NTNU, all of whom are on their final year of a bachelor's degree in informatics. All have different knowledge and expertise within informatics due to different backgrounds and many elective courses in the degree. This results in a very broad field of knowledge, which is useful to the project and can be used to aid in internal knowledge transfers for faster learning and productivity.

1.3 The project goal

The main purpose of the project is to create a working system using a microservice architecture. Due to the project being a demonstration of an architecture, it is important to prioritise displaying the benefits of the architecture, as well as revealing potential disadvantages and possible solutions and workarounds. As such, the CMS itself is not the main focus when shaping the project and when planning. The reason for choosing the CMS as the system to implement was to have something relatable for the employees of TRK when demonstrating microservices.

Chapter 2

Requirements

A monolithic implementation of the system has already been developed for TRK by an external company. The same list of requirements was provided to our group. This list was lengthy and contained several requirements not relevant for demonstrating a microservice architecture. Some of the requirements were thus selected and adjusted in cooperation with the customer, to better suit the project.

The process of streamlining the list of requirements is described in detail below, but the gist of it is that the team would implement a demonstration of the microservice architecture, and to do that, required to focus on a rudimentary version of the final product, which could showcase the advantages and disadvantages of such an architecture.

Throughout the development process the team met with the customer several times. Each time, a demo was showcased, and feedback was given with regards to existing functionality or desired new functionality. This added additional requirements to the initial requirements list throughout the project.

2.1 Selecting requirements

The main criteria for selecting features to implement were based on the project's goal to display advantages and disadvantages of a microservice architecture. This process removed functionality which many deem to be required. However, in an attempt to explore this architecture such functionality would not bring this thesis closer to its goal, therefore they were not prioritised. This includes functionality like deleting an article and signing out of the system. Such features are not required in order to demonstrate the microservice architecture,

and are candidates for being left out in favour of features that more readily demonstrate the architecture and its potential advantages.

In determining which microservices to implement, the requirements from the original document were used as an aid. The requirements which represented key functionality (or otherwise aided in demonstrating the microservice architecture) were aggregated. Through discussion, groups of requirements sharing similar features were chosen, and a list of microservices to implemented emerged, collectively fulfilling the chosen requirements.

It was desirable to show some of the key advantages of using a microservice architecture in addition to the functional requirements. The customer suggested hot-swapping¹ of components, as this reduces downtime in a system if components need to be updated.

2.2 Functional Requirements

The requirements provided by the customer were generalised, and did not suit the development method used in the project (read more about the development method in chapter 3). Because of this, they were re-written as user stories. Since the ratio of user stories to requirements may not be one-to-one, this resulted in more user stories than requirements. This is because user stories are a bit more specialised in order to clarify and remove potential misunderstandings and incorrect assumptions.

Traditionally, user stories consist of three parts: “As a ...”, “I want to ...”, and “so that ...”. In this project, the last part is omitted. This last part can be helpful if the motivation and use of the system is unknown to the team. However, it can also be unnecessary and thus makes the sentences more complicated and confusing than necessary (Cohn, 2008).

Each user story has an ID consisting of an identifier describing which service its for, followed by an integer. This makes it easy to identify which part of the system it refers to when referred to outside of the requirements table (e.g. git commit messages).

2.2.1 Content related services

Content administration was by far the largest part of the project. The main purpose of content administration is to allow employees to create content and publish it on the web. Because of its size, the functionality is divided into three microservices: front-end, publishing, and templates. Their base requirements are summarised in table 2.1.

¹Replace a component without shutting down the system.

Front-end The front-end service is responsible for making the system accessible to users over the web. This includes rendering all of the UI elements and displaying appropriate content to the user. It should also contain an editor that enables employees to create and upload documents. Since it will be used by people with various backgrounds, an important aspect is for the interface to be intuitive and non-technical. For this reason, it is implemented as a WYSIWYG-editor. The editor should have support for templates, making it easy to create standardised content.

This microservice should primarily be focused on the client side of the system. Its server side component should be as minimal as possible, to keep the back-end from becoming too monolithic. The server side component should be there to host the client side part, and to facilitate communication between the client side and other microservices.

Publishing The publishing service is responsible for making content created in the editor accessible to the other microservices. It should allow the other microservices to see what content has been published, and allow them to add or remove content. It should make it possible to set metadata about the content that is published, such as title, description and tags. It is also responsible for where and how content is stored on the back-end servers, and will be uploading the the content files to the servers.

Template The template service is responsible for making templates easily accessible to other microservices, as well as making it possible to manage different templates. These templates should for instance be used by the editor when creating new articles.

Table 2.1: User stories for the content services

ID	Description	Service
CF-1	As a user, I should be able to create articles using a WYSIWYG editor	Front-end
CF-2	As a user, I should be able to add and adjust images	Front-end
CF-3	As a user, I should be able to add videos to articles	Front-end
CF-4	As a user, I should be able to add links to other articles and to web pages	Front-end
CF-5	As a user, I should be able to view articles in an appropriate UI	Front-end
CT-1	As a user, I should be able to choose a template for the article	Template
CP-1	As a user, I should be able to publish created articles	Publishing
CP-2	As a user, I should be able to edit metadata (tags and descriptions) of documents	Publishing

2.2.2 Authentication

The authentication service takes care of authenticating and authorising actions users wish to perform in their tasks. The exact criteria for what it should do is listed in table 2.2. Authentication is done using OAuth2, which allows for a central authentication service which handles authentication and authorisation for all the microservices, rather than an authentication service inside each microservice.

OAuth2 implements token-based authentication, where each token has a lifetime. If a token expires, the user has to re-authenticate. However, as long as the user is active on a service, the user can refresh an active token – being given a fresh token as a result.

In addition to authenticating users, the service also handles authorising users. Some of the content or some of the functions may be access-controlled; this will all be handled by the authentication service in cooperation with the services themselves.

Table 2.2: User stories for the authentication service

ID	Description	Service
A-1	As a user, I want to be able to log in	Auth
A-2	As an administrator, I want to be allowed to create users	Auth
A-3	As an administrator, I want to be allowed to create content	Auth
A-4	The system should allow for different access levels	Auth

2.2.3 Search related services

A single search service which handles all of its functionality could have been implemented, without the service getting too large. As a proof of concept, however, and trying different granularities, the service was split into three services. How the user stories were split amongst them is shown in table 2.3.

Index The index service extracts and stores keywords from published documents, and returns lists of document identifiers as a response to queries for keywords.

Spell checking The spelling service is responsible for providing feedback on query words. It supports correction and completion. Completion returns a list of words which start with the query, correction returns a list of words within a two character edit distance of the query.

Search The search service accepts queries, consults the index service for documents matching keywords in the query, and returns a revised list of documents to the user along with spelling feedback, courtesy of the spelling service.

Table 2.3: User stories for the search, index, and spelling services

ID	Description	Service
SE-1	As a user, I should be able to quickly search for documents	Search
SP-1	As a user, I want suggestions to completion of search terms	Spelling
SP-2	As a user, I want feedback on misspelled words	Spelling
SP-3	As a user, I want autocompletion of words	Spelling

2.2.4 Service-to-service communication

The different services communicate with each other directly by exposing a REST API for the others to consume. The IP addresses of the services are obtained by storing them in a distributed hash table. The DHT is hosted by all the services. This assures that communication between services is not dependent on any single service acting like a broker.

There are no user stories associated with this aspect of the system, as users are not supposed to be exposed to, or aware of such implementation details. It does however have traditional functional requirements, listed in table 2.4.

Table 2.4: Functional requirements for the communication between services

ID	Description	Service
COM-1	The services should be able to communicate with each other	Communication
COM-2	The services should be able to easily query for the IP address of a given service	Communication

2.2.5 Status

To have an overview of each of the services, the customer suggested the implementing of a status service which could provide the state of each service. The team suggested that the service could provide documentation for each service as well, to collect all documentation in one place for easy access. This resulted in two additional user stories for the project, shown in table 2.5.

Table 2.5: Functional requirements for the status service

ID	Description	Service
STA-1	Users should be able to see if each service is available	Status
STA-2	Users should be able to see the documentation for each service	Status

2.3 Non-functional requirements

The non-functional requirements provided were based on the ISO/IEC 25010 standard (ISO/IEC, 2011). This standard comprises eight characteristics, each with multiple sub-characteristics.

The main focus of this project is on performance, compatibility, and maintainability, since these are key points in the demo and architecture. Reliability, security, and portability was set as desired qualities, seeing as these would be important if the system was intended for use beyond demonstration purposes. The last two, functional suitability and usability, were not included at all. This is due to their lack of importance for the demonstration.

Table 2.6 shows a list of the non-functional requirements, where as table 2.7 shows the desired qualities. These are all derived from the ISO/IEC 25010 standard in cooperation with the customer.

Table 2.6: Non-functional requirements

ID	Description	Key attribute
NFR-1	The system should respond within 500ms under normal load (50 req/s)	Performance
NFR-2	The system should be able to handle minimum 750 simultaneous users	Performance
NFR-3	It should be easy to add new instances of a service	Scalability
NFR-4	Services should not have any performance impact on any other services, even when sharing environment	Scalability
NFR-5	The system should divide load between all instances of the services	Scalability
NFR-6	All services should communicate through REST	Modularity
NFR-7	The system should have a minimum of 80% test coverage, to lower the chances of introducing defects or reducing the product quality	Maintainability
NFR-8	The architecture of the system should be well-documented	Documentation
NFR-9	Use of architectural and design patterns should be documented	Documentation

The non-functional requirements mentioned above were created by the customer or by doing some simple research. I.e. NFR-1 was influenced by (Nielsen, 1993) and NFR-2 was introduced by the customer. NFR-3, NFR-4 and NFR-5 were introduced by the microservice architecture, and NFR-6, NFR-7 and NFR-8 were added by the team to supplement the architectural requirements. NFR-9 and NFR-10 were added for documentation and also maintainability purposes.

Table 2.7: Desired Non-functional qualities

ID	Description	Key attribute
NFR-10	The system should handle software faults without going down	Reliability
NFR-11	The system should have a secure authentication system	Security
NFR-12	All services should be easy to install and uninstall	Portability
NFR-13	All services should be easily replaceable with software of similar functionality	Portability

These desired non-functional requirements were requirements which would be “nice to have”, but not required, due to the exploration part of the project. I.e. NFR-10 would be required in a production environment, but for demonstration purposes it should be acceptable to experience some software bugs, as long as the main functionality works as it should. The same applies to NFR-11; a secure authentication system would be required in production, but not all services require this kind of security for exploration purposes. NFR-12 and NFR-13 are connected to the architectural choice.

Chapter 3

Project management

Good planning is essential for a large project to be successful. Decisions like assignment of tasks, sketching of solutions, what tools to use, and when and where to meet has to be decided. An organised plan ensures everybody is on the same page and shares a common understanding of what to do.

Planning can be divided into two parts; planning of the system itself (its architecture and technologies), and organisational planning. The organisational plan is mostly decided upon in early project phases, while system planning is done by individual teams along the way, so to ensure common development practices.

Project management of a microservice project proved to be a new and challenging experience for the team compared to more traditional architectures. Because of the experimental DevOps¹ style of the project and microservice oriented product, it was difficult to maintain a traditional Waterfall or Scrum-like development process. Other development processes like the Spiral model were used instead.

3.1 Risks

During the first group meeting the team did a risk analysis to identify and rank the risks' priorities. There were also made contingency plans for each identified risk. The result is displayed in figure 3.1. A scale from one to nine was used to rate the likelihood and impact of each risk, the importance being given by the product of likelihood and impact. Some risks were added as the project went along, as not all of them were considered initially.

¹DevOps is a movement where the Development and Operations of a system is combined, so that the developers also operate the systems (Loukides, 2012).

There were also some adjustments to likelihood and impact. Doing reevaluations of the risk analysis helped give a better understanding and awareness of the risks in the project, which enabled more easily avoiding and handling them. The risk analysis was weekly reconsidered, either by a few of the members, or the whole team depending on availability. The reevaluation was done by taking new experiences and events into consideration when looking at each of the risks and discussing whether there were any risks missing or in need of adjustment.

Initially, the team had one more member, though a few weeks in this member had to leave the project for personal reasons. A member leaving was not one of the risks considered, though the nature of the project architecture mitigated much of the impact by scaling down on the project scope.

Table 3.1: Risk analysis

Description	Likelihood	Impact	Importance	Preventive action	Remedial action
Members of the team is absent	9	3	27	Divide tasks and plan ahead, so the remaining members do not lack information to continue working	Contact the absent members to obtain the required information. Redistribute tasks
Bad/lack of communication	3	5	15	Regular meetings, team building	Team building
Wrong tools	3	5	15	Do research when new tools are needed and stick to familiar tools	Consider switching to a different tool
Unplanned absence	7	2	14	Keep everyone updated	Redistribute tasks
Unrealistic goals	5	2	10	Use techniques such as estimation poker	Attempt to re-estimate and consider working overtime
Individuals not able to complete task	5	2	10	Inform the group early if you are having trouble	Bring the unfinished tasks to the next sprint, and attempt to finish the task early in the next sprint
Loss of data	1	6	6	Frequent backups and the use of version control	Recreate data
Client can not meet	2	3	6	Give the client multiple options on when to meet	Reschedule as soon as possible
Unclear organisation and responsibilities	2	3	6	Assign clear roles early	Reorganise if possible
The skill of individuals does not match the given task	6	1	6	Map skill/knowledge early	Internal courses, homework readings

3.2 Absence

Over the course of a project this scale, it is to expect that some members are absent or busy for periods of time. Mapping out prolonged absence early helped determine available resources each week. This was important in planning how to distribute the workload and when to implement different parts of the system. For example, from week 10 to 13 half of the team went on an excursion. During this time no major implementations were being developed, and the absent team members were only assigned minor and documentation related tasks.

Figure 3.1 shows the table of when people knew they would be busy. In addition to the written status, the figure was colour coded for easy readability. Each status was defined as follows:

- Green : Available, reply within 8 hours.
- Yellow : Mostly available, reply within 24 hours.
- Bright red : Somewhat busy, will reply when time.

Initially there was also a dark red status, indicating inability to respond until the following week. This was not used, as the team had multiple communication channels and had made arrangements for team members to be at least moderately reachable at all times. More information about the use of communication tools can be found in section 5.5.

Week	Aleksander	Haakon	Håkon	Nicklas	Ole Martin	Pål
5	Available	Available	Somewhat busy	Available	Available	Available
6	Available	Available	Mostly available	Available	Available	Available
7	Available	Available	Somewhat busy	Available	Available	Available
8	Available	Available	Somewhat busy	Available	Available	Available
9	Available	Available	Somewhat busy	Available	Available	Available
10	Available	Available	Mostly available	Available	Mostly available	Mostly available
11	Available	Available	Somewhat busy	Available	Somewhat busy	Somewhat busy
12	Available	Available	Somewhat busy	Available	Somewhat busy	Somewhat busy
13	Available	Available	Mostly available	Available	Mostly available	Mostly available
14	Available	Available	Available	Available	Available	Available
15	Available	Available	Available	Available	Available	Available
16	Available	Available	Available	Available	Available	Available
17	Available	Available	Available	Available	Available	Available
18	Available	Available	Available	Available	Available	Available
19	Available	Mostly available	Available	Available	Available	Available
20	Available	Somewhat busy	Available	Available	Available	Available
21	Available	Mostly available	Available	Available	Available	Available
22	Available	Available	Available	Available	Available	Available

Figure 3.1: Planned absence

3.3 Time frame

Group members were expected to each spend roughly 20 hours per week on the project, as it is a 15 credits course. This resulted in 120 work hours per week at the teams disposal. This was originally 140 hours, though a group member left the team and course early on. Despite this, it was still planned to have 3 teams each working on different tasks throughout most of the project. For some tasks this partitioning turned out to be too small or too big. In those cases, the relevant teams were adjusted accordingly.

Figure 3.2 shows a planned timeline for working on the various tasks.

The deployment deadline was set to 1st of May. This was decided primarily to leave May for writing the report and documentation. It also left sufficient time in case of underestimated time required for development tasks.

Figure 3.2 shows when to implement each service and which are being developed in parallel. Lines 1, 4, and 8 represent report deadlines, lines 2, 3, 6, 7, and 9 represent development deadlines and demonstrations of the system, and line 5 represents the Easter holiday. It is due to its span over the Easter that the search service task is as long as it is.

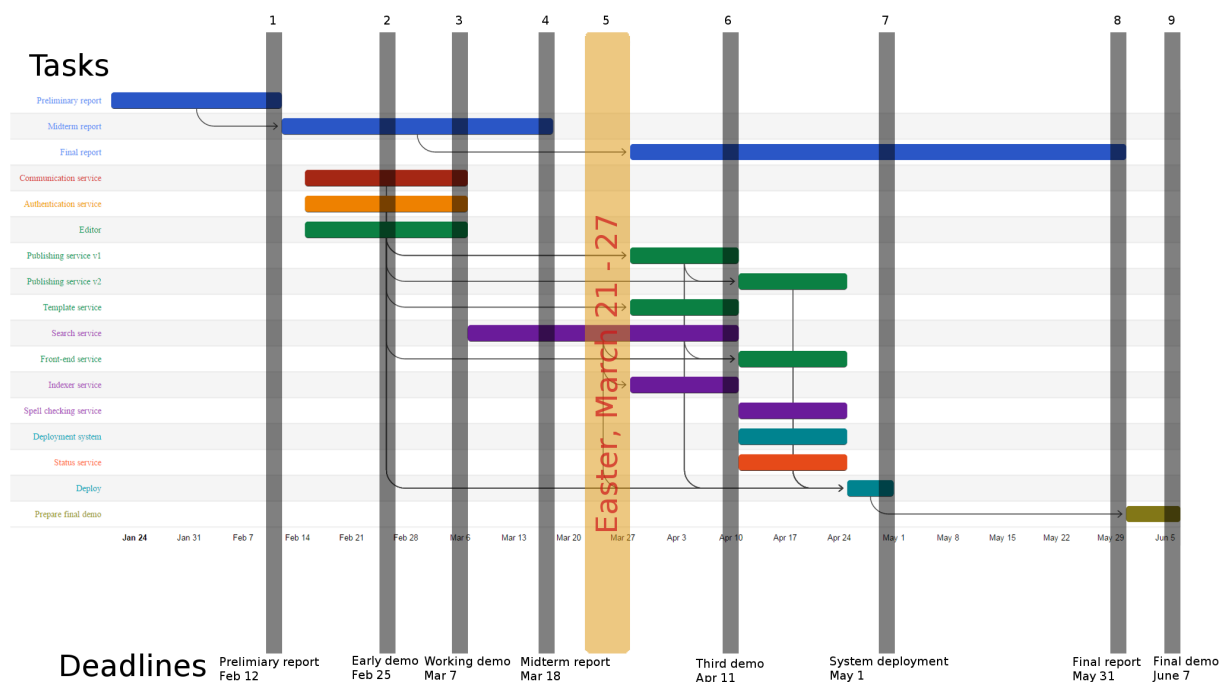


Figure 3.2: Activity plan with tasks and deadlines. Arrows indicates dependencies.

3.4 Estimation

With a lot of focus on researching and exploring advantages, problems, and best practices – estimating each task down to hours was both unpractical and infeasible. Some attempts towards estimating tasks in hours were made early in the project, though was quickly abandoned and story points were to a degree used instead (Martin, 2010). This allows for a more abstract estimate, while not committing to a fixed number of hours. In practice, the story points represented days until launch of the service, one day being about eight work hours. This type of measurement better handles all the uncertainties related to the research needed for each service.

3.5 The team

During the first meeting, all the team members filled out a form with past experiences and language/technology familiarities. This was meant to be an aid for how to pick the members for the sub teams for the different microservices. The results from the meeting are displayed in table 3.2.

Table 3.2: The team, knowledge mapping

Name	Languages/technologies	Experiences
Aleksander Hansen	Python, Java, C#, C++, HTML, JS, LaTeX, MySQL, git	IT1901
Haakon Johansen Jonsson	Python, C/C++, Java, Scala, LaTeX, MySQL, Android, git	Misc. small android projects, IT1901
Håkon Larsen Solbjørg	Python [Django], git, Java, HTML, JS, CSS, REST, LaTeX, UNIX, CI, Testing	dotkom, the startup Islero, IT1901
Nicklas Nilsen	Python, SQL, Fortran, LaTeX, git, UNIX	IT1901
Ole Martin Knurvik	Python, CSS, HTML, JS, Java [Android], LaTeX, C#, git	Design and UI, IT1901
Pål Karlsrud	Python [Django, Flask], Java, C, Perl, Postgres, MySQL, UNIX, Apache, Shell, git, HTML, JS, CSS	IT-Komiteen at Samfundet, IT1901

3.6 Team organisation

The internal teams were assigned and reassigned based on the current requirements and technologies used, so having strictly defined roles throughout the project would have been cumbersome. Although some overarching responsibilities were assigned to some team members, to ease communications and knowing who to talk to regarding e.g. service hosting and operations during testing.

3.6.1 Roles

Group Leader Ole Martin Knurvik

The group leader took care of organisational tasks and delegated work to the other group members. Examples of tasks are booking meeting rooms, communicating with the customer and course staff, et cetera.

CTO Håkon Larsen Solbjørg

The Chief Technology Officer is a role which functions as an advisor. This can be useful when starting developing a feature, for example suggesting a programming language, framework, or library to use, or a sparring partner on how to implement a given functionality.

3.6.2 Responsibilities

UI and UX Ole Martin Knurvik

Has the executive role of design (UI and UX) for the front-end service(s) of the system. This covers following a graphic profile, making sure the design is clean, and that the system is usable from a usability perspective. This role originally included responsibility for usability testing and such, but was omitted seeing as this is not important for an architecture demonstration.

Deployment server(s) (production) Pål Karlsrud

Has the executive responsibility of managing the deployment server(s) that are used in a production environment.

Staging server(s) (development) Pål Karlsrud and Håkon Larsen Solbjørg

Has the executive responsibility of managing the staging server(s) that are used in a development environment.

3.7 Software development process

The overarching guideline of the project was to explore and demonstrate the microservice architecture, rather than an implementation of the product itself. A consequence of this was that Scrum and Kanban would be difficult to strictly follow, as many of the elements

from these methodologies would have seemed like wasted work. This includes elements like Scrum's product backlog, and Kanban's WIP limits. Initially Scrum was considered to be used as the main development process, but it was abandoned in the early phases of the project. With a lot of time dedicated to reading up on microservices, methods, and technologies – following a spiral model was more natural. A spiral model enables using part of the planning phase to do research, and iteratively determine the objectives and what functionality to implement (Boehm, 2000).

This is not to say no elements from other methodologies were used. Daily stand-up meetings were used to keep everyone up to date on how the others were doing, allowing them to come with suggestions and feedback when needed. The objectives of each iteration could also be compared to the sprint backlog in Scrum, or To Do in Kanban.

As shown in figure 3.3, there are four main phases of each iteration. Each iteration started with research, as gathering knowledge about best practices, advantages, disadvantages, and customer preferences was essential in determining the next set of objectives – in this case which functionalities to implement – to best demonstrate microservices.

In step two of the iteration, determining the objectives, the client was often included and encouraged to come with feedback based on ideas and thoughts derived from the research. Once the objectives were established, the risks needed to be mapped out and resolved. In the fourth and final step, development and testing, the objectives were naturally implemented and tested before deployment. Due to the small size of each service this was either done alone or with a partner. In this part, techniques like pair programming or Kanban were options for use. This decision was left up to each team depending on what they felt more comfortable with.

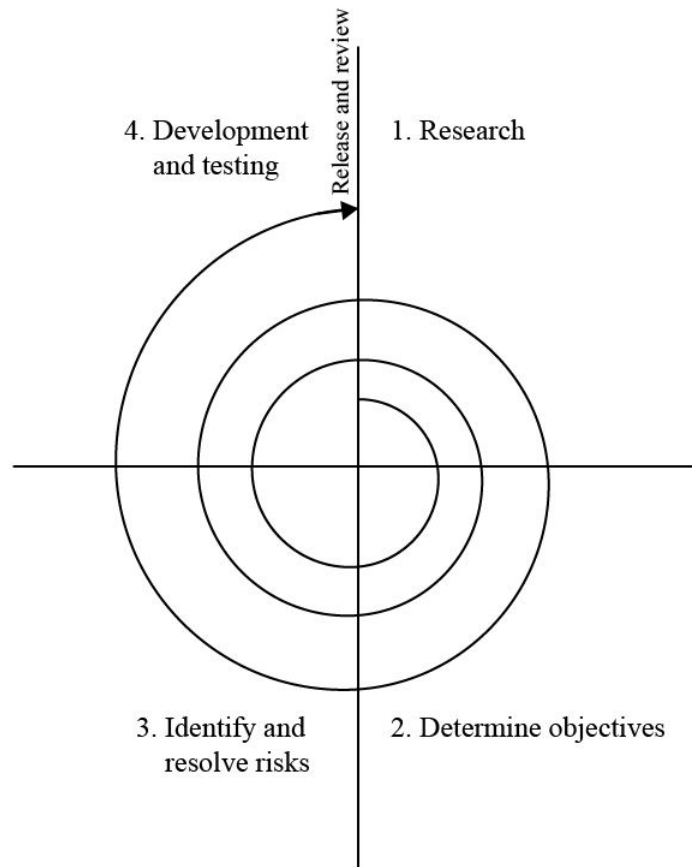


Figure 3.3: Development method

3.8 Iterations

The length of each iteration has been a bit fluid and varied from one to three weeks. This due to the huge differences in size and uncertainty of each service implementation. Seeing as the team was split into groups which did not depend on each other, apart from the deadlines and deliveries, there was no reason why each group should be locked to the same length for each iteration.

3.8.1 Iteration 1

21st of February – 7th of March

During the first iteration, the focus was on three parts of the system: the editor, the authentication service, and the communication service.

The group was divided as follows:

Editor

- Haakon Johansen Jonsson
- Ole Martin Knurvik

Authentication

- Håkon Larsen Solbjørg
- Nicklas Grimstad Nilsen

Communication

- Pål Karlsrud
- Aleksander Hansen

The goal of the iteration was to get a working prototype up and running for a demonstration with the customer the 7th of March. This prototype was to consist of an editor, which was to communicate with the authentication service. This meant that a user should be able to log in and then use the editor. The user stories that cover this functionality is shown in table 3.3.

Table 3.3: User stories for iteration 1

ID	Description	Service	Story points
CF-1	As a user, I should be able to create articles using a WYSIWYG editor	Front-end	8
CF-2	As a user, I should be able to add and adjust images	Front-end	3
CF-3	As a user, I should be able to add videos to articles	Front-end	3
CF-4	As a user, I should be able to add links to other articles and to web pages	Front-end	2
A-1	As a user, I want to be able to log in	Auth	8
A-4	The system should allow for different access levels	Auth	8
COM-1	The services should be able to communicate with each other	Communication	13
COM-2	The services should be able to easily query for the IP address of a given service	Communication	3

Most of the iteration was spent on learning and choosing which technologies to use for the various services. This took a lot more time than originally anticipated, as the team had little experience with most of the technologies that were planned for use. This motivated the switch from Scrum to the spiral model.

Combining the services proved to be harder than expected. While all the services worked by themselves at the end of the iteration, communication between the editor and the authentication service was not ready for the demonstration.

During the course of the iteration it was decided, in agreement with the customer, that a distributed communication system would be used. A WYSIWYG-type editor was chosen, and templates were to be fully editable. “Embedded code” was interpreted to mean embedded objects like video. The requirements as they were before any changes were made is found in appendix A.

3.8.2 Iteration 2

28th of March – 11th of April

Between the 7th and 28th of March, only the search service was in development; the next proper iteration did not start before the 28th of March. This was due to Easter, having to write on the report, and a large part of the group being unavailable.

The second iteration focused on the first publishing service, the template service, the search service, and the index service.

The group was divided as follows:

Publishing service

- Haakon Johansen Jonsson
- Ole Martin Knurvik

Template service

- Pål Karlsrud
- Håkon Larsen Solbjørg

search and indexer service

- Nicklas Grimstad Nilsen
- Aleksander Hansen

The goal of this iteration was to create an improved prototype, capable of demonstrating publishing and viewing of articles, searching, and loading templates. The user stories that cover the functionality in this iteration is shown in table 3.4.

Table 3.4: User stories for iteration 2

ID	Description	Service	Story points
CF-5	As a user, I should be able to view articles in an appropriate UI	Front-end	3
CT-1	As a user, I should be able to choose a template for the article	Template	13
CP-1	As a user, I should be able to publish created articles	Publishing	8
CP-2	As a user, I should be able to edit metadata (tags and descriptions) of documents	Publishing	2
SE-1	As a user, I should be able to quickly search for documents	Search	13

The improved prototype did have the features that were planned to be implemented in this iteration. The template and search services were not connected to the front-end, and had to be demonstrated by themselves. The publishing service however could demonstrate communication between services.

The original plan was to have every microservice supply their own user interface, and have the client-side load them individually. During the iteration it was decided that there would be a separate front-end service to deal with the user interface, because communication within the client-side of the project turned out to be really difficult. After the demonstration for the customer, an additional requirement – having two implementations of the publishing service – was added by request from the customer. It was also decided that features like calendar information, attachments, and support for different file types in the search service, were not important to the goal of the project, so the related requirements were removed.

Chapter 4

Alternative solutions

Following the microservice architecture with loosely coupled services, there are unbounded options and combinations of solutions for the frameworks alone. Many of the technologies chosen for the final solution were based on previous experience and knowledge, and some on their compatibilities with other solutions. This chapter discusses some of the choices that were made and alternatives that were considered.

4.1 Technologies

The customer expressed a preference for microservices being implemented using a variety of technologies, so that the modularity- and separability benefits of microservices would become more apparent. Beyond that, the customer had no preferences for any specific technologies. The back-end of the search microservice was created in Python, because of familiarity with the language, as well as Python having a sizable set of libraries with the required functionalities. Python was not a necessity, however – the team also had familiarity with Java and appropriate Java libraries – so Java was an option as well.

Searching and indexing was implemented as two separate services. The main purpose of this decision was to further demonstrate the modularity of microservices; it would however be possible to replace the two services with a new conjoined service which provided both searching and indexing. An example of an alternative solution would be to use Apache Solr, which provides a superset of the functionality implemented by the current solution (Karusell, 2011).

Similar technology decisions were applicable for the other microservices as well, enabling

a mixed composition of technologies to showcase the large degree of decoupling microservices posit.

4.1.1 Deployment alternatives

How the different services are deployed is especially important when working with microservices, as one of the goals with microservices is for it to be easy to add or remove instances of a service. The following alternatives for deployment of services were considered: using a single server for all services, creating a VM for each service, and creating a Docker container for each service.

At first running all the services on the same server was considered. The problem with this is that the services would have been harder to distinguish, as they may run several similar processes (Cohen, 2015). Another problem is that this would make it relatively hard to scale to multiple servers, since all servers would need to run all services.

Running all the services on the same server would also provide very little isolation between the services. Lack of isolation might cause problems when using microservices, as it is common to use many different technologies which provide some of the same functions, and this might cause the technologies to interfere with each other.

Using VMs for each of the services solves many of the problems above. A disadvantage with VMs is that they have high overhead, as they usually run a full operating system. VMs do however provide more isolation than with Docker containers or running the services directly on a server (Seshachala, 2014), though at the cost of more overhead.

Not needing quite the isolation that VMs provide, Docker containers was chosen for deployment. While being more lightweight than VMs, Docker also makes it easy to deploy custom, sandboxed, self-contained environments for each of the services, which means that a specific microservice can be started and stopped or replaced, without affecting other services (Seshachala, 2014).

The Docker containers are run on a Debian server. Debian was chosen for its stability and high level of documentation. Windows, Ubuntu, or Fedora were options as well, but Debian has more server related documentation, and is more minimalistic than Ubuntu (Mikoluk, 2013). Each Docker container uses Alpine Linux, as this provides a minimal operating system while still providing all the services and packages required (Gliderlabs, 2016).

4.1.2 API structure

Multiple services needing to communicate introduced the need for well-defined APIs. A service accesses the functionality of a target service through the target's API. For instance, a service responsible for hosting content might require clients to have specific permissions, necessitating a service for authenticating users and proving the authentication state of users.

REST APIs exposing JSON data was chosen as the underlying interface technology, and the DRY-principle for communication between the various services, although SOAP and WSDL are viable alternatives for defining and exposing the functionalities of the various services through XML-formatted data rather than JSON.

4.1.3 Service-to-service communication alternatives

The way the microservices communicate is a very important part of the microservice architecture, as it largely determines how loosely coupled the services will be (PwC, 2014).

The following three communication architectures were considered: having the services communicate directly, using a message-broker, and using a fully distributed architecture. Tables 4.1, 4.2, and 4.3 summarise the pros and cons of these architectures.

Table 4.1: Pros and cons of direct communication

Pros	Cons
Simple	Potential single point of failure
Easy to scale	Requires some form of service discovery and registry
Easy to monitor	

Direct communication (illustrated in figure 4.1) was considered early on. In the illustration, service B and C wishes to communicate with service A. For this to be possible, service B and C each need the IP address of service A – so to enable the service-to-service transmission of content.

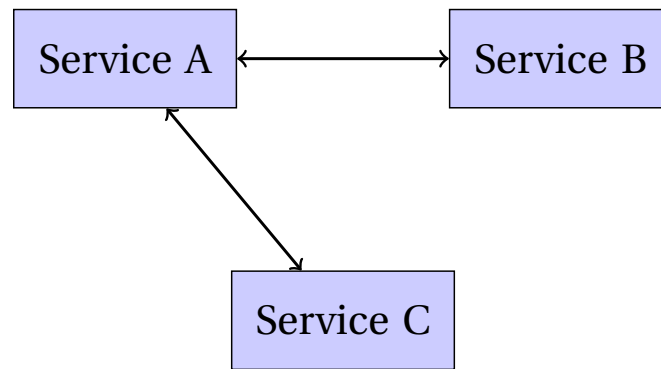


Figure 4.1: Example of communication using direct communication.

The plan was to use an automated service discovery program called Registrator (Labs, 2016), along with the key/value store features of Consul (Mammutus, 2015). The advantage of this approach would be that to obtain the IP address of a service, a service need only know how to query Consul. The role of Registrator would have been to automatically register services using Consul.

However, Registrator turned out to not be as suitable as it at first seemed, since it lacked proper support for versioning of APIs. Registrator also lacked features for making services available only after the service had actually started up – this is important, as a service might use some time to start, during which it might not be ready to accept requests.

In order to find a solution which could easily be modified to fit the needs of the system, a message-broker architecture was considered. This would solve the notification problem by having each service register itself, and periodically send heartbeat messages to the broker, to signal that the service is still online. A message broker architecture would also remove the need for direct communication, and instead allow the services to communicate using channels provided by the broker (Richardson, 2015a).

Table 4.2: Pros and cons of message brokers

Pros	Cons
Fast	Potentially single point of failure
The clients only need to know about the relevant channels	Hard to change message format
Scales well with multiple services	Must have relevant bindings for language used in microservice

Figure 4.2 illustrates how the services would communicate with each when using a message broker. Using the message broker architecture, service B and C would send a message to the broker in order to communicate with A. The broker would function similarly to a router, since it routes messages between services.

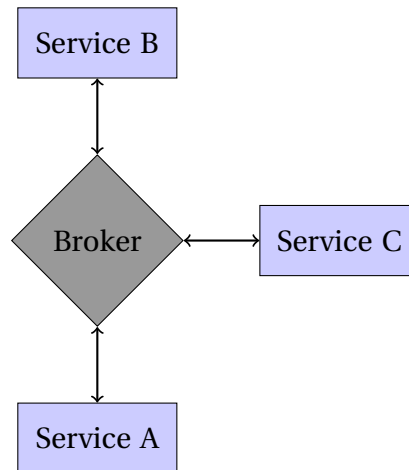


Figure 4.2: Illustration of communication using a message broker.

For larger networking loads, communication using message brokers tends to be easier to scale than direct communication, by enabling asynchronous communication. This results in endpoints being able to offload the responsibility of transporting messages (Richardson, 2015a).

This did however require writing a solution from scratch, which proved more difficult than it first seemed, particularly because large parts of the message handling had to be written from the bottom-up, and none of the frameworks that were looked into, such as ZeroMQ and RabbitMQ, would allow easily creating the messaging pattern shown in figure 4.2.

Further consideration resulted in an architecture using DHT to store the IP addresses for each service. This is illustrated in figure 4.3. In this architecture, each of the nodes in the DHT collectively store the IP addresses associated with each service, and when a service wishes to connect to another service, it need only query the DHT to obtain an IP address.

Table 4.3: Pros and cons of a fully distributed system

Pros	Cons
No single point of failure	Complex
Relatively easy to query	Takes some time before a node is marked as dead
	Makes changes to the communication system harder to implement.

This alternative eliminates the single point of failure, and each of the nodes used to store information in the network can easily be replaced without bringing down the entire network. cursory research did not result in finding any other microservice projects having used this approach, making it an interesting solution to explore.

Figure 4.3 illustrates how the DHT architecture works. Service B and C first need to request the IP address to an instance of service A, and then communicate directly with service A using the received IP address.

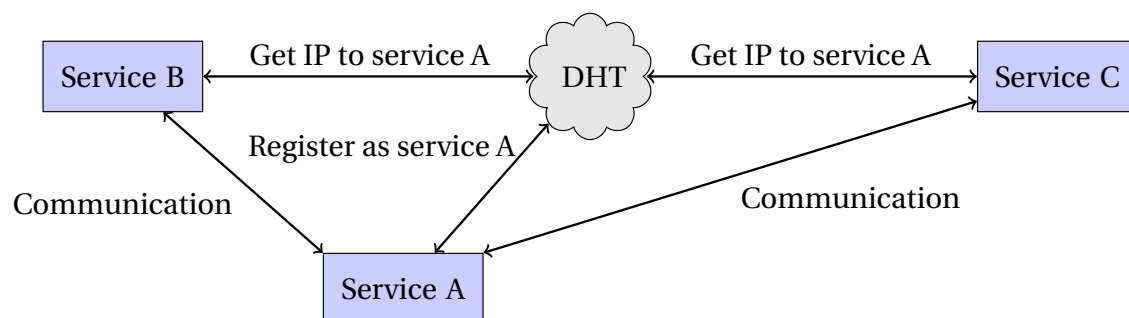


Figure 4.3: Communication between services using the DHT architecture.

4.2 Development method alternatives

A spiral model based development method was chosen (as explained in section 3.7). Alternatives like Scrum, Kanban and Waterfall were discussed, though all fell a bit short. The requirements were likely to change, and services were likely be added or removed, so following a waterfall method was considered impractical. This is because Waterfall requires a lot of planning early in the project, when in the case of this project all the information needed was not yet available.

A solution implementing Kanban or Scrum could have worked, but seeing as how much

the requirements and functionality was changing, having a strict backlog and organising it would require much effort, while giving few benefits (Uzility, 2014). Also many of the team members were inexperienced in using agile development or the artifacts they come with.

Chapter 5

Tools and technologies

This chapter describes the tools and technologies used. This includes development tools and frameworks used, like version control systems, programming languages, and third party frameworks and libraries.

5.1 Development tools

5.1.1 Git

Git was chosen for version control, with GitHub as the repository hosting service. Git was chosen as it makes it easy to manage code, and every team member had experience using it.

5.1.2 Go

The Go programming language was used for the microservice which hosts templates. Go was primarily chosen because it makes concurrent operations easy, which makes it simpler to make a high performance service (Krill, 2015).

5.1.3 Python

Python was used for most of the microservices, as it was the language most of the group members was experienced in. One of the main advantages with Python is that it enables

quick prototyping, and at the same time tends to result in manageable and modular code. Several different Python frameworks were used for the microservices.

Django web framework was used for the front-end and one of the publishing services, since it is well established with good documentation, a lot of supported packages and is actively developed. Also, some team members already had experience with it, and it seemed fairly easy to learn for those familiar with Python.

Twisted is an event-driven asynchronous networking engine for Python. Twisted was chosen for several of the services because it is quick to set up and configure, without a lot of web-server boilerplate (twistedmatrix.com, 2016).

Entangled is a DHT and tuple space based on Kademlia written in Python (Aucamp). The project was forked from its GitHub repository and used as a starting point for the communication back-end.

5.1.4 JavaScript

JavaScript was used for client side scripting and for the server side of one of the publishing services.

Summernote is a JavaScript library for creating simple WYSIWYG editors (team). It was chosen as the basis for the editor because it is not very complex, it creates a simple user interface, it is fairly easy to modify if needed, and it still covers our requirements.

One of the publishing services is written in JavaScript to demonstrate that the microservices can be written in different languages and still be compatible and interchangeable with each other. JavaScript was chosen to achieve diversity without using a great deal of unfamiliar languages.

Node.js was used as the JavaScript runtime environment with the Express.js web framework. This was because they seemed to be the most established and documented tools for their respective tasks.

5.1.5 MongoDB

MongoDB is a document database. It was chosen for the publishing and templates services because their task is primarily to store and retrieve article and template documents, which is what a document-oriented database like MongoDB is designed for (Inc). MongoDB is also easy to work with, since it stores documents in a JSON-like format, and data is usually formatted as JSON when it is transferred between microservices. This was also a good opportunity for some group members to learn about NoSQL databases.

5.1.6 PostgreSQL

PostgreSQL is an open source database system. It was chosen for the indexing service. PostgreSQL was chosen as it provides a combination of stability and performance (Group, 2016).

5.2 Deployment tools

The following subsections describe the tools that are in use to facilitate an easier deployment phase. All the microservices run containerised behind a reverse HTTP proxy, and all logs are shipped to a central repository. Due to the DevOps methodology, all services are continuously deployed, and therefore continuous integration tools were required.

5.2.1 Docker

Docker is a tool that spawns an application in a containerised environment. It behaves much like a virtual machine, but it shares several of the base components with the host. By doing so, Docker provides isolation between different containers, while remaining lightweight. It has a simple command line for managing containers, which makes it very easy add or remove new services, and add new instances of existing services.

Virtual machines abstract away the hardware components of the host machine and emulate their own. This makes it possible to run the same system almost anywhere, with the same components – even though the host machine is different.

5.2.2 Docker Compose

Docker Compose is a command-line interface for managing multi-container Docker applications, and makes it easy to scale and upgrade the system by providing commands for these specific actions (Docker, 2016).

5.2.3 Nginx

Nginx is a an open source web server. This provides a common interface for load balancing and routing from the Internet to the different microservices.

5.2.4 Teamcity

TeamCity is a CI system. It is provided for free and runs tests for each service upon a version control check in, giving feedback on whether tests pass. CI and testing is described in chapter 7.

5.2.5 Jenkins

Jenkins is a CI system, which was used to automatically deploy the services. Jenkins was chosen as it provides several plug-ins for deploying Docker containers.

5.2.6 Travis CI

Travis is another CI system. It is distributed and runs in the cloud rather than on premise, which allows for very simple test-running for small systems. Travis was chosen for the service used to serve templates, as the service was written in Go, which is incompatible with TeamCity, as well as requiring special accessories not integrated in TeamCity by default.

5.2.7 Logstash

Since every service is run in a containerised environment, an application to centralise the log files produced by each service was needed – otherwise the container of each service would have to be accessed to obtain the log files. Logstash was used to solve this problem. Logstash

is an application which makes it easy to centralise log files produced by multiple servers and make them accessible in a shared format (Smith, 2015).

5.3 Testing

For testing, the libraries associated with the specific programming languages was primarily used, as specialised libraries are more flexible than forcing all tests into a specific framework. When releasing new versions of the microservices, all the tests are run using a CI system. This ensures that all tests pass before the new version is actually released. Read more about testing strategies in chapter 7.

5.4 Project management tools

Seeing as git was already being used for code, all the other documents could have been synced through it as well. However, for real-time collaboration on the same files, real-time editing tools like Google Drive and ShareLaTeX were used instead.

Google Drive was used for all project related documents, such as plans, meeting summaries, and illustrations. Google Drive was chosen due to its real-time collaboration for creating and editing a wide variety of files, in addition to simply uploading and sharing most other file types.

The reason why Google Drive was chosen, is that it has excellent integration with the Google Docs system, which makes it easy to edit and create new documents.

ShareLaTeX was used for easy collaboration when writing the report in LaTeX. It is also significantly more user friendly than the normal LaTeX toolchain, and allows collaborative real-time editing. The report could have been edited locally, and then synchronised using git via GitHub, but this would most likely cause many merge conflicts and would make it cumbersome to write the report.

5.5 Communication tools

The group members, the client, and the supervisor all had very different time schedules. This necessitated some sort of asynchronous online communication platform.

E-Mail was chosen as the main communication platform with both the client and the supervisor outside the meetings. E-mail enables participants to answer when they have time to spare, without having the expectation to always reply immediately that more instant messaging applications do.

Slack was chosen as the main communication tool within the group. Slack allows for creating chat groups for different topics (e.g. Report writing, testing, general). A plugin for slack called Geekbot was also used. Geekbot is a stand-up bot, which each day asks all members of the team what they achieved yesterday, what they plan on doing today and what (if anything) is impeding their progress (VentureGeeks, 2016). This requires less commitment than physical stand-up meetings, and is especially helpful when the team is not able to meet on a daily basis.

Chapter 6

Architecture

The architecture, and especially communication between services, is important when creating microservices, as this largely determines the maintainability, efficiency, and modularity of the system. Choosing a suiting architecture is therefore very important.

6.1 System architecture

When building a microservice based system, it is important to consider the granularity of the services (Bloomberg, 2016). A coarse grained interface was chosen, i.e. each service receives and sends JSON structured information. The goal of each service is to have a well-defined function and that each microservice consists of components that are easy to reuse. By making each service have a well-defined function, a loose-coupling between the services can be maintained.

Without loose coupling, modifying a service often necessitates modifying all services that interacts with it. This can become a problem in monolithic and to an extent service-oriented architectures, because of their unclear guidelines for separating services (Clark, 2016).

This system consists of eight microservices. They are described in section 6.3, and the architecture shown in figure 6.1. One of the services has two implementations, in two different programming languages, in order to show how a service could easily be replaced in the microservice architecture. The status service is not shown in figure 6.1, as it performs monitoring, and is not part of the CMS system.

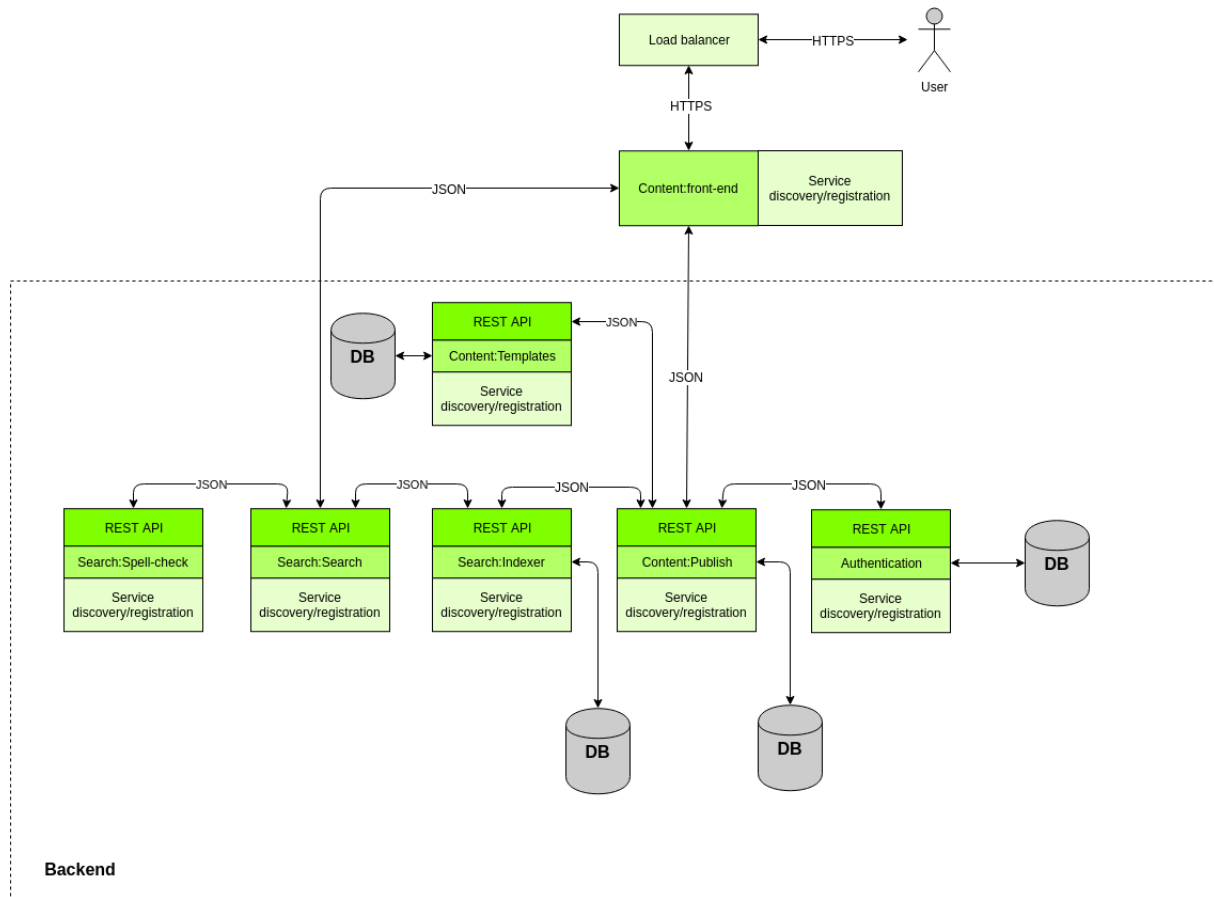


Figure 6.1: Connections between different microservices and other components.

6.2 How the services communicate

A DHT works by distributing the responsibility for maintaining the mapping from keys to values among the nodes in the network. As mentioned in section 4.1.3, this property is exploited to store the type of service and the IP addresses associated with that type. This also automatically distributes the load among all nodes, and by doing so fulfils NFR-5.

The node is divided into two parts: one part for communicating with the other nodes in the network, and one part which exposes a REST API to the service, so that it can easily query the network for IP addresses.

This removes the need for creating language specific bindings, and makes the node easier to test, as HTTP requests are sufficient to obtain data from a given node. Each service runs a node as a part of the microservice.

When a service *A* wants to communicate with another service of type *B*, it begins by querying the node to obtain an IP address to a node *C* of type *B*. This allows service *A* to communicate directly with *C*. If *A* detects that *C* does not respond, it simply removes *C* from the list of available services. This flow is illustrated in figure 6.2

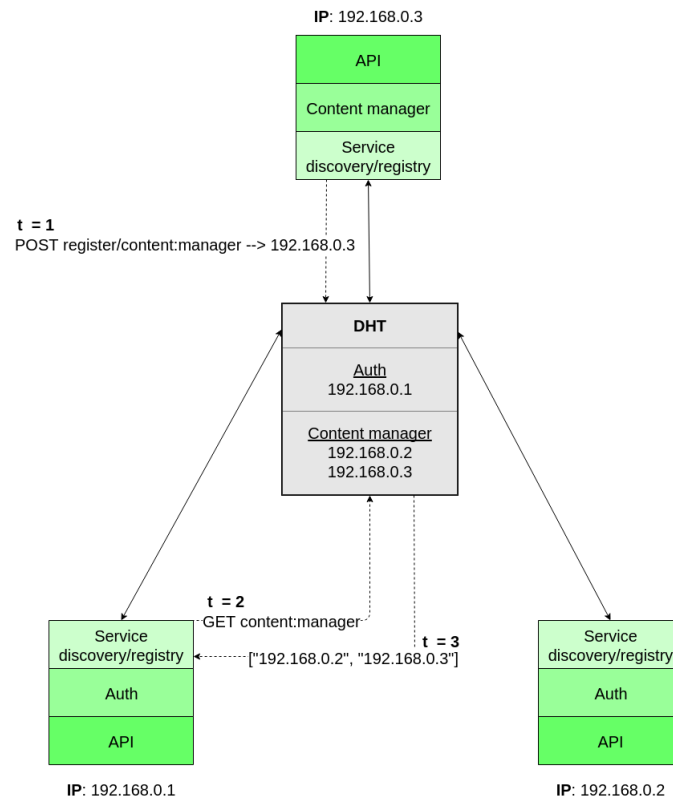


Figure 6.2: Flow of requests during a query.

6.3 The architecture of each microservice

Originally there were five microservices related to the creation and publishing of content: editor, publishing, datasets, administration, and template service. These would form the core of the system. Figure 6.3 shows an overview of their architecture.

It was however decided that the datasets and administration services would be unnecessary for the purpose of demonstrating the microservice architecture, so they were excluded.

It was also decided that having all the microservices supply their own front-end would be cumbersome and unnecessary, because many of them had very small or no front-end at all. So instead, all the front-ends were merged into a unified front-end service. The editor turned out to not need its own back-end at all, so it was entirely merged into the front-end service. An overview of the new content architecture is shown in 6.4.

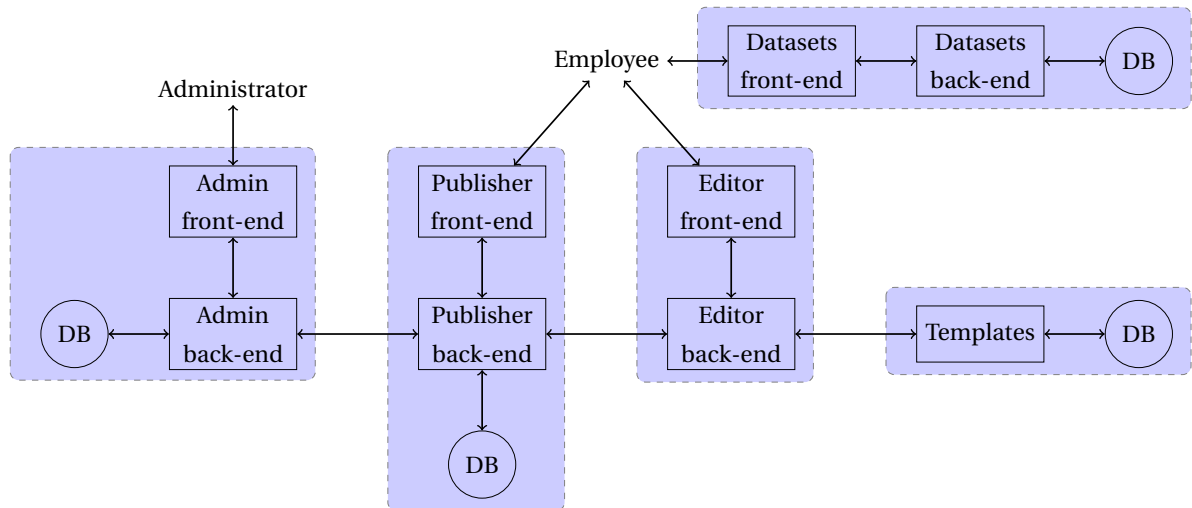


Figure 6.3: Overview of the original design of the content services.

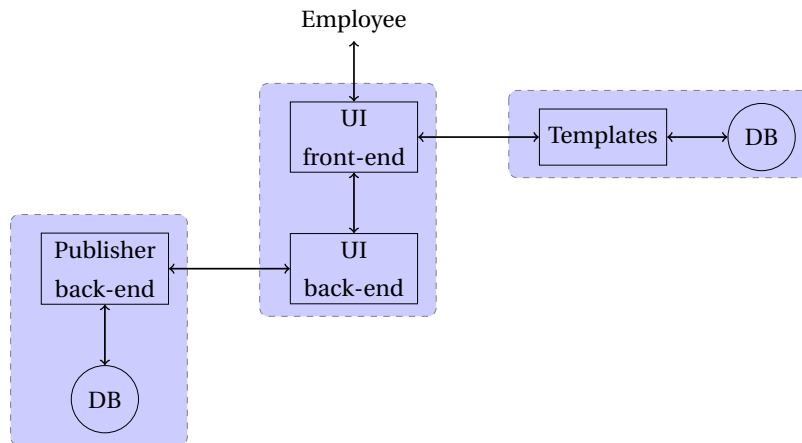


Figure 6.4: Overview of the refactored design of the content services.

The search microservice was also changed. Originally, it was supposed to be a single microservice – but the team responsible for implementing it decided to split it into three microservices, as it would otherwise have to be larger than first anticipated.

6.3.1 Front-end service

The front-end of the project uses Django to render the pages according to templates. This service works by asking other services for content and displaying it to the user in its proper template. By having one service render all of the visible content, it is easy to change the overall look and feel without having to change more than one service. It also uses some JavaScript to implement Summernote as the WYSIWYG editor.

6.3.2 Publishing service

The publishing service provides an HTTP API to store and retrieve articles and information about articles, and to get a list of currently stored articles. It works by assigning a unique identifier to articles when they are inserted, which can then be used to fetch or delete articles. Each article is stored as a JSON object with fields for the article body and various information about the article.

The Publishing service is implemented twice. One implementation is written in using Node.js and the express.js web framework, the other is written in Python using the Django web framework. Both implementations use MongoDB to store the articles, since a simple key/value store for mapping IDs to JSON objects is all that is needed.

6.3.3 Template service

In order to make it easy to add, remove, or get a template, a dedicated service for template management was created. This service is written in Go, and uses MongoDB to store templates. MongoDB was chosen because a key/value store was sufficient, since only the name of the template and the HTML associated with each template was going to be stored.

The service provides an API which can be used to obtain and manage the templates in the database. The API was implemented using the web framework library Gin, which makes creation of web applications easy.

6.3.4 Status services

To have an overview of the status of each service, a status service was made. The service uses HTTP to try to get a specific resource on one of the services, to verify that the service responds to queries. In addition, it links to the documentation for the service.

6.3.5 Authentication service

Microauth is the service used as the authentication service. It is made using Python and Django, implementing OAuth2. The service wraps around an implementation of OAuth2, and handles registering, authentication, and authorisation by providing an API for communication. Any client will communicate through the OAuth2 provider either directly or indi-

rectly, by passing tokens as part of their requests. The tokens are mapped to users and will therefore make sure to provide whatever content is available to the users.

There are also tokens set up between the services themselves, as confidential tokens, which connects and authenticates the services between each other – so they know they can trust each other, therefore preventing fraudulent services from entering the network as a man in the middle.

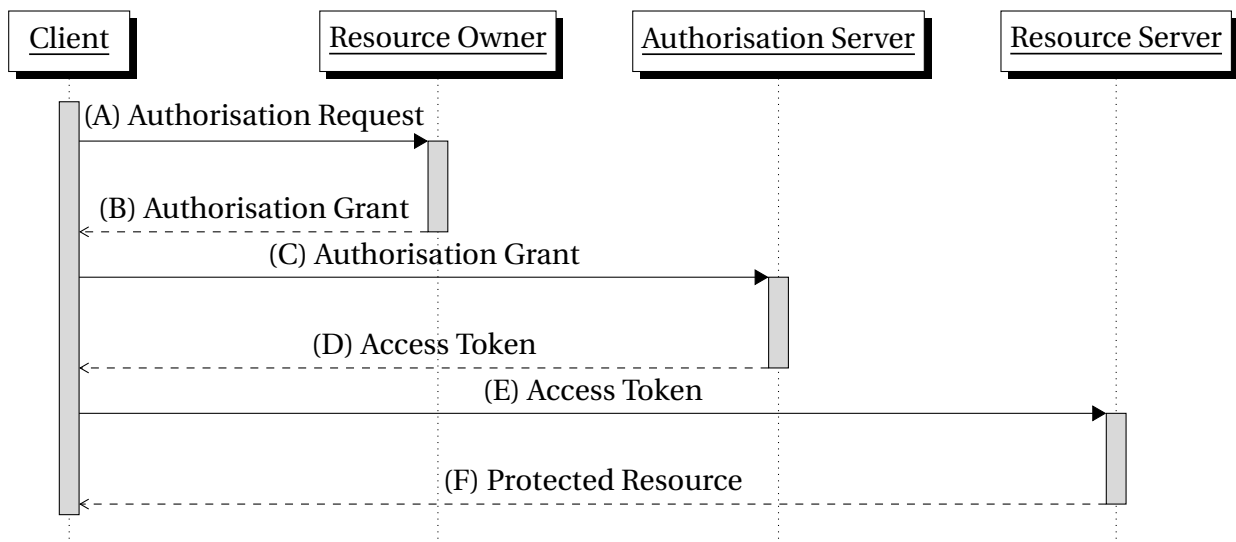


Figure 6.5: The flow of OAuth2 is as described in the OAuth2 RFC (Hardt, 2012), section 1.2.

6.3.6 Index service

The index service provides keyword based retrieval of documents. It is notified by the publishing service whenever an article is published or modified, and updates its mapping appropriately. When queried with a keyword, the service returns a list of articles containing it. The index service also provides additional information about keywords, such as frequencies and completion lists.

The index service is implemented in Python, and uses the Twisted networking engine (twistedmatrix.com, 2016).

6.3.7 Spelling service

The spelling service is queried with an individual word. It has two functions – spelling correction and word completion. If the query concerns spelling correction, the service consults

a frequency word list and uses edit distances to find possible candidates for correction. If the query concerns completion of the query word – for instance in the use case of typing queries in search boxes, where the last word in the query is only partially completed – the service consults its word list, returning possible completions ranked by frequency. Both generic and content-specific spelling feedback is supported, through the use of multiple frequency lists.

The spelling service is implemented in Python, using the networking library Twisted. It also uses a probabilistic model from Norvig for finding words within specific edit distances of the query (Norvig, 2007).

6.3.8 Search service

The search service's area of responsibility is to return lists of documents to users, as a response to search queries. Users make requests through a search front-end, which conveys queries to the search back-end via its exposed API.

The search back-end communicates with the index and with the spelling service. The index service is queried for articles matching keywords in the user-supplied query, while the spelling service is queried for feedback to the user on potentially mistyped or incomplete words.

The back-end is implemented in Python, using the networking library Twisted. The Python Natural Language Toolkit (Steven Bird and Loper, 2009) is used for stemming and ignoring stopwords, which have minimal value for searches.

Figure 6.6 shows the sequence of calls during a normal search. The user submits a query through the search front-end to the search back-end, and the index is consulted for a list of matching documents for each keyword in the query. The search back-end may then retrieve and rank the documents in the list from the index.

The back-end also consults the spelling service for spelling corrections, the spelling service constructing a list of candidates for each keyword in the query not recognised as a valid word. The search back-end returns both the results and the list of spelling feedback.

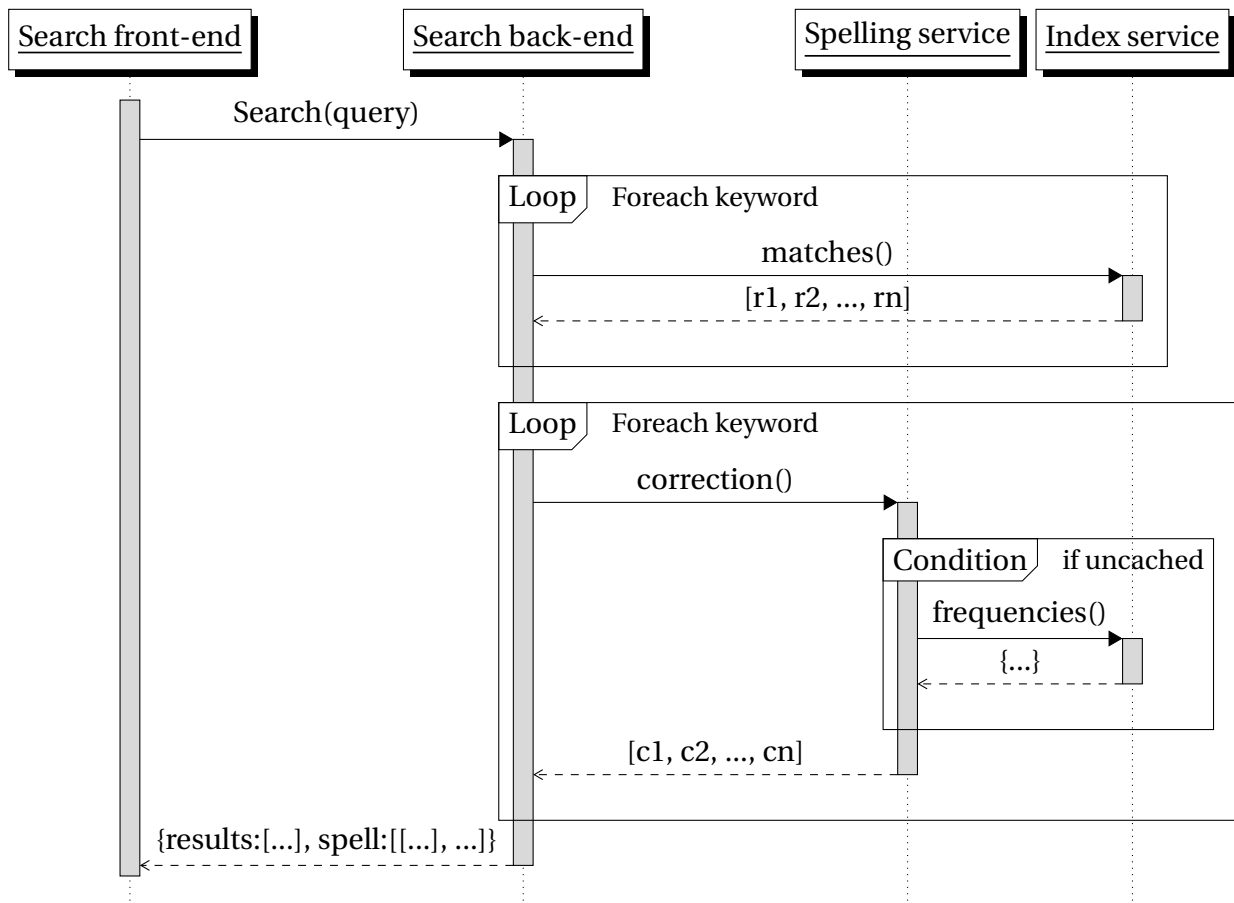


Figure 6.6: Sequence diagram of a complete search query. A list of results and a list of lists of spelling feedback (indexed by position in query sentence) is returned.

Not shown is the sequence of calls for partial searches; partial searches do not return any results beyond spelling feedback. For partial searches, the sub-list corresponding to the positionally last word in the query is a list of possible keyword completions rather than corrections. The remaining sub-lists are the usual corrections.

6.4 How the architecture scales

Figure 6.7 illustrates how adding more instances of the front-end service improves the overall response time of the service, thus showing how the system scales. The service was tested using Siege, which is a HTTP load testing and benchmarking tool (Fulmer, 2012). Siege was used to simulate 250 simultaneous users which requested the frontpage every second for 15 seconds. In order to see whether the response times were consistent, Siege was ran three times so that the data from each run could be compared. The server used to run the system was a DigitalOcean droplet with 4GB of RAM and two CPU cores.

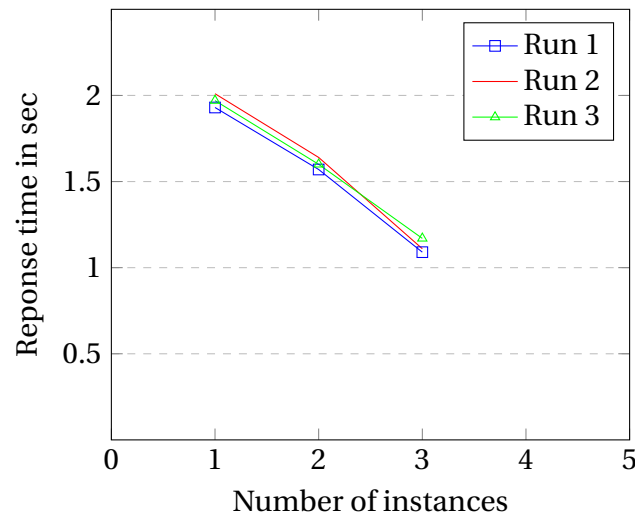


Figure 6.7: The scalability of the front-end service

Figure 6.8 shows how the publishing services scales when adding more instances. The same exact setup as above was also used to test this service. As figure 6.7 and 6.8 shows, the system scales relatively linearly.

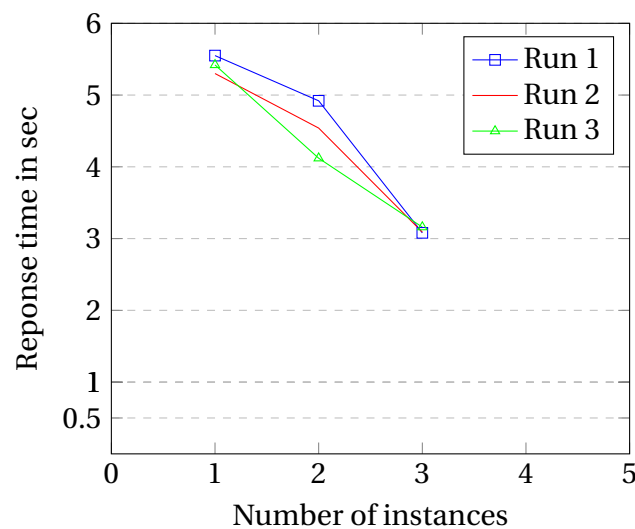


Figure 6.8: The scalability of the publishing service

The performance of the system is significantly better when dealing with 100 users making 50 RPS, which according to non-functional requirement NFR-1 is the normal load. Figure 6.9 shows how one instance of the front-end service performs under normal load. The average response time is 0.25 seconds, which fulfils NFR-1. Siege was used to simulate 100 simultaneous users making requests every 1-3 seconds for one minute, which is equivalent to a load of approximately 50 RPS.

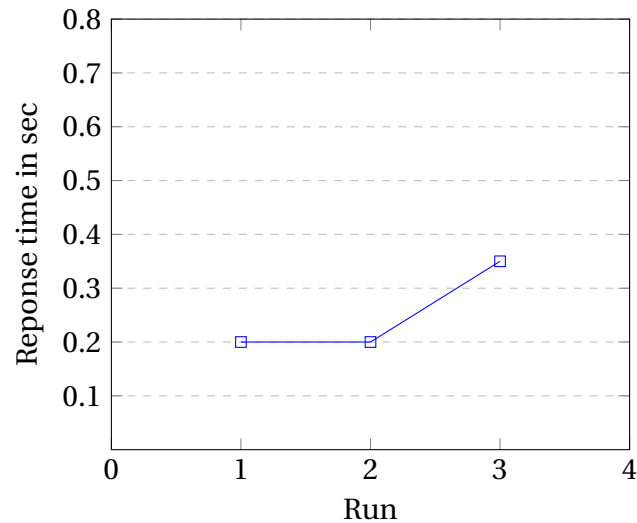


Figure 6.9: The performance of the front-end service under normal load

6.4.1 Stress testing

Figure 6.10 shows how the system handles an increasing amount of users. The response time increases linearly with the amount of users. Siege was used to simulate the users. Each user requested the site every 1-3 seconds for 30 seconds. The delay at 750 users is noticeable, but it would be easy to add new instances of the service in order to reduce the response time.

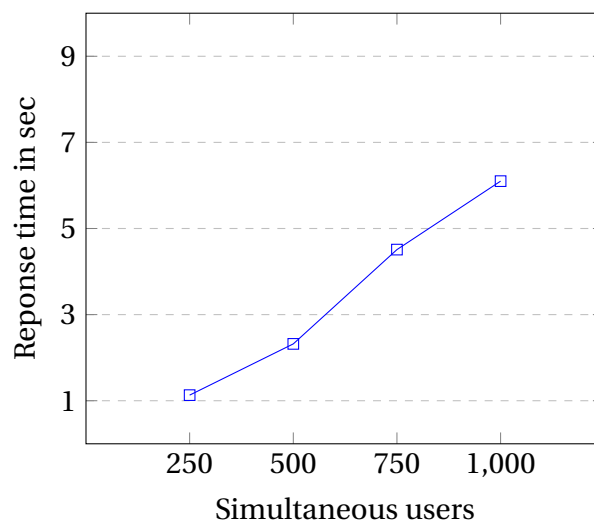


Figure 6.10: How the front-end service performs under stress

6.5 Deployment

Each instance of a microservice is deployed in a Docker container. This makes it easy to add new instances of each microservice, and services can be upgraded by simply replacing the container. Management of containers is also made easy by creating pre-packaged images that can be managed through Docker Compose. Since each container has separate log files, Logstash is used to collect and store logs. Figure 6.11 shows the overview of how the services are deployed.

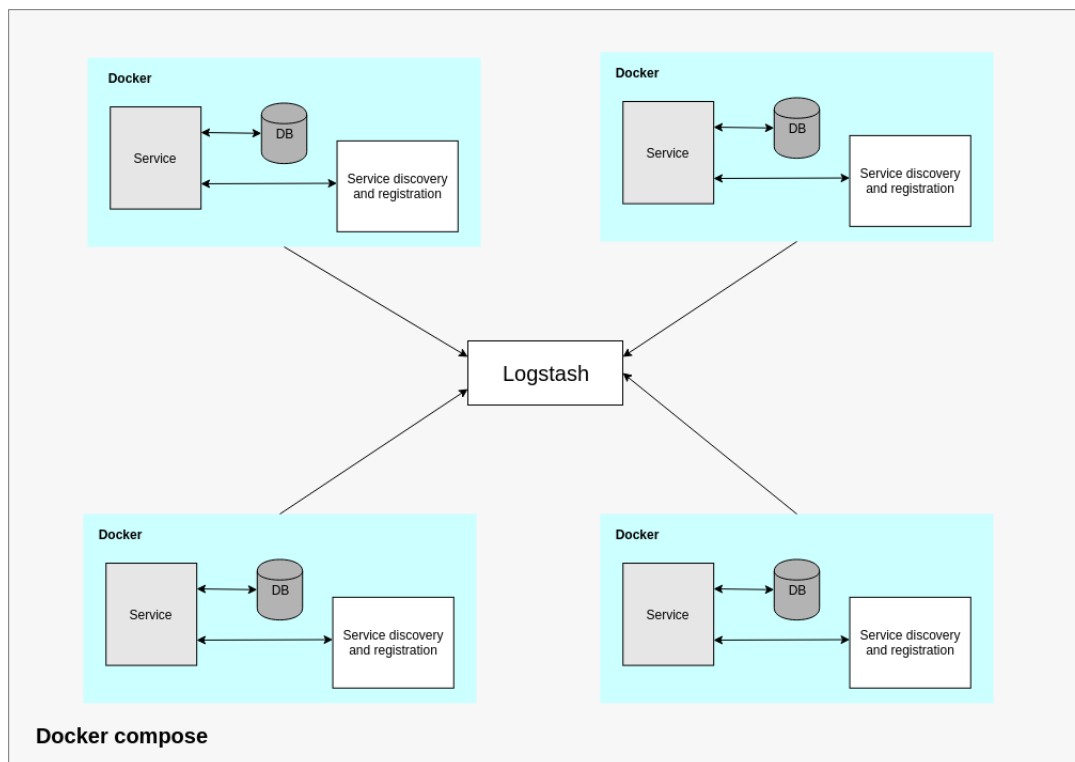


Figure 6.11: Deployment of services.

6.6 How the system could be improved

Several ways of improving the system was discovered while developing the system. This section describes some of the ideas on how to improve and how they would improve the system.

6.6.1 Architecture

The primary reason for choosing to use a DHT to perform service discovery, was to experiment with an alternative way of performing service discovery. We would not recommend doing this in a production environment, as this is not as mature of a communication architecture as many other service-discovery architectures, it lacks a proper way of recovering from errors, and has little support for health checking.

The implemented method also adds unnecessary complexity to each service, since the entire service has to be updated if the program running the DHT needs to be changed.

We would therefore instead recommend using a different tool such as Consul for this, which is a distributed key-value store. It is fault tolerant and performs extensive health checks. It can also be distributed over multiple servers to avoid a single point of failure (Mammutus, 2015). Consul works similar to the DHT used in this project, i.e. by providing a REST API that can be used to obtain the IP address. This means that this system easily could be modified to use Consul instead.

6.6.2 Deployment

One big problem when working with microservices is determining how the data is to be duplicated across the services of the same type. For instance, if one service is used to host articles, how will services of the same type receive articles posted to another service than itself? This was solved by hosting a database separately, and having all the services of a given type connect to the same database.

A better solution would however be to host the databases internally in each service, and having the databases function as a cluster. This would provide redundancy, as well as flexibility through enabling the use of different versions of the same database. It would most likely also lead to a more efficient system, as the data would be stored locally accessible with each service instance. By using clustering the system would improve performance, but it might not be fully up-to-date on data in other clusters. This is a problem that occurs for any clustered system, so the implementation would need to take this into account to achieve data-consistency.

In a larger-scale environment, it would also be a good idea to use a cluster management application such as Mesosphere. Applications like mesosphere are built for handling large-scale microservices and containers (Anicas, 2014).

6.6.3 Services

The front-end was implemented as one service, in order to avoid too much complexity and maintain changeability. One argument for why the front-end should have been split into at least two different services is the fact that if this had been done in a full scale project, one could argue that the architecture of the front-end is more a SOA than it is a microservice architecture.

One of the ways a true microservice architecture could have been achieved even with a larger scale project, would be to have an Angular or React based front-end¹, rather than Django. This would allow each service to expose its own front-end through its API. This way, the services would depend less on each other and there would not be any single point of failure.

¹AngularJS and ReactJS are front-end JavaScript frameworks used for interactive web pages.

Chapter 7

Testing

Testing is a vital part of any software product. The term “testing” is quite broad, as it has to do with the customer’s satisfaction of the product, as well as the end users and other stakeholders. This includes automated tests like unit tests, testing non-functional requirements such as a maximum required response time from the system, and an acceptance test according to the requirements as a whole. In other words, the process of testing is to make sure that some product meets its criteria.

For this project, the customer voiced opinions regarding following the DevOps movement. Due to this, testing becomes an even more vital part, as any release of any service will be tested mostly by automated tools and can be released at any time. This requires very high code coverage so that any wrongdoings will be caught before they reach production.

7.1 Test strategy

This test strategy follows the Heuristic Test Strategy Model (Bach, 2015). This model helps designing a test strategy by defining a set of patterns to follow. Its purpose is to remind testers about what to think about while testing, while it also gives a non-technical overview in some sections. This will facilitate discussion within the group with regards to testing, to achieve better test practices and a better knowledge of what to test and how. The strategy describes the general approach to the testing process, in which it informs the team about how to approach testing with regards to customer’s, environments, elements, and qualities.

7.1.1 Project environment

The customer of the project do not have any end users in mind, as it is a demonstration of an architecture. Therefore, the customer is the sole stakeholder and expectations are rather low with regards to functionality, but high in terms of technology.

Previous implementations of the microservice architecture, like the one talked about in (Richardson, 2015b), have been a heavy influence on this project.

The services are tested by expecting them to supply output given some specific input. In a microservice architecture, it is important that the services respond to expected queries, therefore this should suffice as integration testing. Each service also has unit tests to make sure their internal logic functions correctly.

7.1.2 Product elements

The end product lives in containers (as described in section 5.2.1), so the hardware of the system is well-defined and tested already. Therefore, the focus of testing the services points towards internal logic, data handling, input and output, and its interfaces. These tests also cover the operations of the product.

7.1.3 Quality criteria

The quality criteria being tested are the ones defined in section 1.1 and 2.3, namely performance, maintainability, modularity and scalability.

Performance, modularity and scalability are important qualities for the end product with regards to how the system operates and therefore the usability for the end user. Maintainability is an important aspect of the microservice architecture and is therefore defined as a quality criteria to be tested.

7.1.4 Test techniques

The test techniques for this project can be split into automated tests and manual tests. *Function* and *domain* testing was done automatically (unit tests, regression tests and integration tests run by TeamCity), while *flow*, *stress* and *user* testing was done manually during development.

Flow testing was done during development of each module, making sure that doing the steps in any order would produce relevant feedback, and that doing them in the correct order would produce the desired results.

Stress testing is described in section 6.4.1.

User testing was also done during development and in cooperation with the customer during demonstrations, by showing the product and allowing them to give feedback regarding functions and behaviour.

7.2 Testing microservices

Since the project followed the DevOps movement, testing became a vital part of the project. As described in section 5.3, each service mostly used their own unit testing framework for testing. However, with multiple programming languages and also multiple frameworks, having a person with executive responsibility for testing was not up to par due to insufficient knowledge about the different languages. The CTO stepped in for some tips and tricks, but the teams mostly stuck to testing based on the requirements for the project.

7.3 Automated tests

Automated tests are tests that can be completed without any human interaction. This includes, but is not limited to, unit testing and integration testing. These are tests that execute some code and then expect some functionality by mocking a case and verifying the response.

As described in subsection 5.2.4 and subsection 5.2.6, the test suites for this project runs automatically on Travis CI and TeamCity upon a VCS check-in. All test runs of the project are available at Travis CI¹.

¹<https://travis-ci.org/microserv/>

7.3.1 Continuous Integration

Continuous Integration is the process of running automated test suites whenever a change is released, to make sure that the new change does not break existing functionality, or rather, that new features do not alter existing functionality in a way that breaks the current implementation. The word “integration” takes into consideration multiple modules of the system by testing their integration with each other.

7.3.2 Continuous Deployment

Continuous deployment is the process of deploying some system continuously, like with CI, whenever a change is released. Continuous deployment requires CI to make sure that the recent release passes all tests and the product can safely be released. This does not account for downtime of the end system, which might happen when releasing.

7.3.3 Types of tests to automate

Regression tests are tests that should uncover problems between new code and current code. This means that if a new implementation breaks an existing test, either the new code does something unexpected to the existing code, or the new code does something *expected* to existing code, but the tests are not updated to handle this.

Code coverage is an automated way to check how much of a code base its tests cover. Some code is *covered* by tests if that code is executed during the testing process. This gives an estimation of how much of the project is covered by tests, and therefore how “safely” new code can be considered ready to deployment. However, executing a line of code does not mean all branching factors are covered. Code coverage gives a decent estimate, but considering different possible paths of execution is important as well.

All code coverage stats of the project is available at Coveralls².

At the time of last deployment, the coverage stats for the services are as follows:

²<https://coveralls.io/github/microserv>

Table 7.1: Code coverage statistics for the various services

Service	Code Coverage %
Frontend	79 %
Microauth	74 %
Templates	33 %
Search	93 %
Spell-check	84 %
<i>Average</i>	<i>73 %</i>

Disclaimer: The *average* calculation does not take into account the size of the service, it is merely a total average over all of the services.

As per NFR-7 in subsection 2.6, the system should have a code coverage of above 80 %. According to the numbers in table 7.1, the project does not achieve this goal. Most services are above, but some are below. This was mostly due to time limitations, in which unfamiliarity with testing and testing practices was a key problem.

7.4 Testing of third party software

A key point when choosing frameworks and libraries for the project services has been choosing recognised, well-tested and documented open source software. Other key points are community/general activity around the software (so that it is not outdated code no-one uses or maintains anymore), and a non-restrictive license.

Since it is ascertained that the third party frameworks used are well-tested, it would be redundant to retest their behaviour in these services. Custom implementations and wrappers of the frameworks are tested, but the frameworks and their test suites are trusted to be good enough for general use in this project.

7.5 Testing non-functional requirements

The non-functional requirements defined in tables 2.6 and 2.7 have been tested by simulations described in the following paragraphs and by selecting fitting architectural patterns. By having a distributed (containerised) system, a lot of unforeseen problems can be handled – like hardware and software/firmware failures.

Performance is tested by stress-testing the system, as described in subsections 6.4.1 and 7.1.4.

Compatibility is achieved through simple guidelines for developing in the project, namely each service exposing a REST API.

Maintainability is tested by running automated tasks for testing, readability, and code coverage – as well as making sure that all services are documented.

7.6 Acceptance test

The final release of the project will be presented for the customer 7th of June in TRK's offices in Trondheim. Considering this is after the deadline of the project, the team scheduled an acceptance test of a release candidate the 28th of April.

The main focus of the project has been to explore microservices as an architecture. Therefore, the acceptance test made sure that the project complied with its requirements defined in chapter 2. Since the solution implemented a couple of new technologies like the OAuth2 flow (described in subsection 6.3.5), the acceptance test also introduced and questioned the flow of the system.

The testing process took place at 13:30 on the 28th of April with the customer (Anvaari Mohsen) present. The acceptance test passed with no problems to mention and no changes to the code.

After the test sequence, the customer voiced interest in sketching an agenda for the demonstration and presentation of the project in their offices on 7th of June.

Chapter 8

Summary and Conclusions

8.1 Architecture

8.1.1 Architectural advantages

In developing microservices, separation of responsibilities as a guiding principle has resulted in benefits like reuse of code and modularity. This is a notable advantage over other architectures, such as Server-Client and Service Oriented Architecture. Server-Client architectures are generally monolithic, and over time in a production environment they grow unwieldy as new features are added, and old features are hard to replace without extensive code refactoring, because of internal backwards-compatibility.

Service Oriented Architecture is an older architecture than microservices, though is in a way similar. Nevertheless, SOA has unclear and varying definitions and the architecture does not have the same overall guideline of separation of responsibilities as the microservice architecture. This leads to SOA often growing monolithic, as the responsibilities of each service grows, rather than being split into more services like with the microservice architecture.

8.1.2 Architectural disadvantages

A complication with the microservice architecture is the separation of responsibilities. The group spent a while deciding which microservices to implement, and even throughout the project decisions were changed, as with the search service.

Microservices, having large degrees of separation, also incur some communication overhead when using the exposed functionalities of each other. A related problem to this is how to share information that is common among services, without having redundancies. The group decided to give each service that needed one its own database, even though this incurred penalties with regards to redundancy. This was to ensure modularity and not having a service being replaced affect other services usage of stored data to a large extent.

8.2 Problems and solutions

8.2.1 Time schedule

Figure 8.1 illustrates a typical Wednesday from 8:00 to 16:00, where a check mark indicates the team member being available, while unchecked indicates unavailable. Each row represents a team member. Most of the days in a week, including weekends, looked somewhat like this. The combined time schedules made it very hard to have regular meetings. Monday and Thursday afternoon were the only options which suited the entire group throughout most of the semester. Therefore all meetings with the client and supervisor were scheduled for these time slots.

Wednesday								
08:00	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00
✓	✓	✓	✓	✓	✓	✓	✓	✓
		✓						
✓	✓	✓	✓	✓	✓	✓	✓	✓
					✓	✓	✓	
✓	✓	✓	✓	✓	✓	✓	✓	✓
✓	✓					✓	✓	✓
		✓						

Figure 8.1: Example of our schedule. One row for each team member.

8.2.2 Abroad

One of the group members had a part-time job in London the first half of the semester. This did not help the already troublesome schedule crash. In addition, there was an excursion in March that three of the members were attending. This all meant a way of having stand-up meetings and exchanging information more frequently than the two meetings a week was needed. Geekbot was chosen for arranging online stand-up meetings. Communication is detailed in section 5.5.

8.2.3 Group member leaving

At the beginning of the second iteration, a group member had to leave the course and the project for personal reasons. This meant that the group would go from seven people, down to six. The group worked around this by reassigning the workload to the remaining members. This announcement happened at a less unfortunate time, as the tasks for the second sprint had not yet been fully decided upon; planning for a six person group was relatively easy.

This could of course have become problematic later on in the project if the total workload was larger than the reduced team could handle. Keeping in mind that the customer was more interested in a demonstration than the product features, scaling down functional requirements accordingly did not concern the customer much.

8.3 Lessons learnt

8.3.1 Project organisation

Schedules Matching schedules is not a necessity for a project to succeed, but it makes the job a whole lot easier. Through planning and different tools, the problems were simplified, but with more similar schedules this could have been avoided in the first place.

Development method Looking back at the project, starting out with a more defined development method would have been beneficial and saved us some time. Agile development methods like Scrum work better when the team has a common background regarding development tools and how to use them. Digital Scrum boards, stand-up meetings, and other

elements take time to learn and accustom to, and thus often benefit less than the required time investment for small projects.

Meetings Even though it was not a huge problem, having more frequent meetings and work sessions would have improved the overall progression of the project early on.

8.3.2 Development

Tools and frameworks Learning to use tools and frameworks not familiar to the team may take more time than first anticipated. Thus, one should try to mostly use tools people are familiar with, unless the tool is required and there are no familiar alternatives.

Status The status service could contain a cache of documentation and check if a newer version is available, to make sure documentation for a service is available even if the service is unavailable.

Front-end The implemented front-end of the system is a bit too tightly coupled to be a true microservice. By splitting it up, or having different services provide their own front-end, this could have been achieved. This would require more work, despite being a nice possible addition to the project.

8.4 Conclusion

The project started with some difficulties due to absence of one of the team members. This was handled by selecting an appropriate tool to facilitate team communication. This turned out to work quite well and had no major impact on the work flow. The roles and responsibilities of each team member were settled in a natural way, as each team member took on tasks that had been prioritised by the team. As the project evolved the different sub-teams worked on their part of the project and reported on the status during the weekly meetings. During the meetings people had the opportunity to share information and seek help on any problems encountered.

Communication with the customer has been good overall, even though some more preparations could have been done beforehand by the team. All in all the project went smoothly and no major problems were encountered.

Even though the system is only a demonstration of microservices, it is a possibility that a third party might be interested in continuing working on the project. Therefore, steps have been taken to ensure that future developers can easily continue on the project if needed. The API for every applicable microservice has been documented, so the services are easy to work with. If creating a replacement for a microservice is desirable, it is easy to find out which features require being implemented. A setup guide has been created to make it easy to deploy a new instance of the system. There are also tests for the services that can be used to validate future changes and make sure the system remains stable.

8.5 Summary

The project originated in the customer's expressed desire for a system implementation using microservices. The group, through this project having implemented such a system, has to a large degree noticed many of the advantages that microservices are purported to have, which reflects many of the customer's hopes and expectations. Overall, we feel that the requirements and goals outlined have been achieved to a satisfactory degree.

Bibliography

Anicas, M. (2014). *An Introduction to Mesosphere*.

<https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere>.
[Accessed 2016-05-14].

Anvaari, M. (2016). Personal communication (E-mail).

Aucamp, F. *Entangled*. <http://entangled.sourceforge.net/>. [Accessed 2016-05-26].

Bach, J. (2015). Heuristic test strategy model. <http://www.satisfice.com/tools/htsm.pdf>. [Accessed 2016-05-19].

Bloomberg, J. (2016). *Think big with microservices: Lego-like software development takes shape*. <http://techbeacon.com/think-big-microservices-lego-software-development-takes-shape>. [Accessed 2016-02-09].

Boehm, B. (2000). Spiral development: experience, principles, and refinements. <http://www.sei.cmu.edu/reports/00sr008.pdf>. [Accessed 2016-05-20].

Clark, K. J. (2016). Microservices, soa, and apis: Friends or enemies? http://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html. [Accessed 2016-05-30].

Cohen, U. (2015). *Docker...Containers, Microservices and Orchestrating the Whole Symphony*. <https://dzone.com/articles/dockercontainers-microservices>. [Accessed 2016-05-09].

Cohn, M. (2008). *Advantages of the "As a user, I want" user story*. <http://www.mountingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template>. [Accessed 2016-05-09].

Docker (2016). *Overview of Docker Compose*. <https://docs.docker.com/compose/overview/>. [Accessed 2016-05-29].

Fulmer, J. (2012). *Siege Home*. <https://www.joedog.org/siege-home/>. [Accessed 2016-05-29].

- Gliderlabs (2016). *Alpine Linux Docker Image*. <http://gliderlabs.viewdocs.io/docker-alpine/>. [Accessed 2016-05-11].
- Group, P. G. D. (2016). *Advantages*. <http://www.postgresql.org/about/advantages/>. [Accessed 2016-05-19].
- Gundelsby, J. H. (2014). *Erfaringer fra 150 mikrotjenester fordeler og ulemper*. <http://www.slideshare.net/janhenrik2/erfaringer-fra-150-mikrotjenester-fordeler-og-ulemper>. [Accessed 2016-05-10].
- Hardt, D. (2012). *The OAuth 2.0 Authorization Framework*. <https://tools.ietf.org/html/rfc6749>. [Accessed 2016-02-24].
- Inc, M. *MongoDB*. <https://www.mongodb.com/document-databases>. [Accessed 2016-05-26].
- ISO/IEC (2011). *ISO/IEC 25010 - Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models*.
- Karussell, P. (2011). *Apache Solr: Get Started, Get Excited!*. <https://dzone.com/articles/apache-solr-get-started-get>. [Accessed 2016-05-14].
- Krill, P. (2015). *Google's Go language is off to a great start, but still has work ahead*. <http://www.infoworld.com/article/2896575/google-go/googles-go-language-pros-and-cons.html>. [Accessed 2016-05-19].
- Labs, G. (2016). *Service registry bridge for Docker*. <http://gliderlabs.com/registrator/latest/>. [Accessed 2016-05-14].
- Loukides, M. (2012). *What is devops?* <http://radar.oreilly.com/2012/06/what-is-devops.html>. [Accessed 2016-05-27].
- Mammutus (2015). *Microservice Service Discovery with Consul*. <http://www.mammatustech.com/Microservice-Service-Discovery-with-Consul>. [Accessed 2016-05-14].
- Martin, J. (2010). *Agile Chalk Talk: Story Points* [video file]. <https://www.youtube.com/watch?v=90Xx8QVnXRc>. [Accessed 2016-05-09].
- Mikoluk, K. (2013). *Debian vs. Ubuntu: Which is the Right Linux Distribution for You?* <https://blog.udemy.com/debian-vs-ubuntu/>. [Accessed 2016-05-09].
- Nielsen, J. (1993). *Response time: The 3 important limits*.

- <https://www.nngroup.com/articles/response-times-3-important-limits/>. [Accessed 2016-05-28].
- Norvig, P. (2007). *How to Write a Spelling Corrector*. <http://norvig.com/spell-correct.html>. [Accessed 2016-03-14].
- PwC (2014). *Rethinking integration: microservices*. <http://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/features/microservices.html>. [Accessed 2016-05-11].
- Richardson, C. (2015a). *Building Microservices: Inter-Process Communication in a Microservices Architecture*. <https://www.nginx.com/blog/building-microservices-inter-process-communication/>. [Accessed 2016-05-09].
- Richardson, C. (2015b). *Building Microservices: Using an API Gateway*. <https://www.nginx.com/blog/building-microservices-using-an-api-gateway/>. [Accessed 2016-05-09].
- Seshachala, S. (2014). *Docker vs VMs*. <http://devops.com/2014/11/24/docker-vs-vms/>. [Accessed 2016-05-11].
- Smith, L. (2015). *Why Logstash is the backbone of the ELK stack*. <http://blog.logit.io/why-logstash-is-the-backbone-of-the-elk-stack/>. [Accessed 2016-05-14].
- Steven Bird, E. K. and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media Inc., <http://nltk.org/book>, 2nd edition.
- team, S. *summernote*. <http://summernote.org/>. [Accessed 2016-05-26].
- twistedmatrix.com (09.02.2016). *Twisted*. <http://twistedmatrix.com/trac/>. [Accessed 2016-05-18].
- Uzility (2014). *Introduction to Scrum - 7 Minutes* [video file]. <https://www.youtube.com/watch?v=9TycLR0TqFA>. [Accessed 2016-02-19].
- VentureGeeks (2016). *Geekbot*. <https://geekbot.io/>. [Accessed 2016-05-09].

Appendix A

Requirements

A.1 Content module

Table A.1: Functional requirements for the content services

ID	Description	Service
CA-1	All content should be referred to with permalinks	Content:site-administration
CA-2	The admin should easily be able to change menus. Both adding and removing elements	Content:site-administration
CA-3	The system should display content based on the users role	Content:site-administration
CD-1	The system should make information from webservice available when creating content	Content:datasets
CD-2	The admin should be able to publish calendar information, downloadable for google calendar and MS	Content:datasets
CE-1	The system should have a WYSIWYG-editor for creating and updating content	Content:editor
CE-2	The admin should easily be able to add and adjust images	Content:editor
CE-3	The admin should be able to add attachments to pages	Content:editor
CE-4	An admin should be able to add embedded code to content pages	Content:editor
CP-1	The admin should be able to publish text, pictures and files to webpages (homepages, intranett, Min Side)	Content:publication
CP-2	The system should structure content hierarchically	Content:publication
CP-3	The system should allow for hiding, and hiding/deleting content after a timeframe	Content:publication
CP-4	The admin should be able to edit metadata (add synonyms, content type, keywords, description, tags and change publisher, language etc)	Content:publication
CP-5	The admin should be able to disable indexing for specific content	Content:publication
CP-6	The user should be able to create links to other sub-sites/articles etc.	Content:publication
CP-7	The system should be able to automatically render information obtained from a rest API to an appropriate UI	Content:publication

A.2 Authentication module

Table A.2: Functional requirements for the authentication service

ID	Description	Service
A-1	Both admins and users should be able to log in	Auth
A-2	Administrators should be allowed to create users	Auth
A-3	Administrators should be able to hide and unhide content	Auth
A-4	An admin should be able to decide which auth groups has access to the different pages	Auth

A.3 Search module

Table A.3: Functional requirements for the search service

ID	Description Service	
SB-1	The system should provide a fast search functionality	Search:Back-end
SB-2	Content should be indexable by external search engines, but an admin should be able to disable this manually	Search:Back-end
SB-3	The system should provide search functionality like stemming and other search enhancements	Search:Back-end
SB-4	The search engine should provide indexing of documents, like PDF, DOC, XLS, etc	Search:Back-end
SF-1	The search engine should provide suggestions to completion of the search terms	Search:Front-end
SF-2	The system should correct commonly misspelled words and names	Search:Front-end
SF-3	The search engine should provide autocomplete of words	Search:Front-end

Appendix B

Setup guide

Initial requirements

- A server running Debian (Jessie or newer) with more than 2GB of RAM and 10GB of free space. The server should also have git installed.
- A domain pointing to the server mentioned above.
- SSL/TLS certificates for the domain mentioned above SSL/TLS is not required, but it is recommend as you would otherwise have to change some of the services. ¹
- Prior experience with Linux and the command line.

Step 1 - Installing the required software

Begin by installing docker, docker-compose and nginx, which is required to run the system.. If the server is running Debian Jessie, you must first add the Debian Stretch package repository, as Nginx 1.10 or newer is required.

¹Certificates can be obtained for free by following: <https://www.digitalocean.com/community/tutorials/how-to-secure-nginx-with-let-s-encrypt-on-ubuntu-14-04>

The Stretch package repository can be used by adding:

```
deb http://http.debian.org/debian stretch main
```

to `/etc/apt/sources.list`, and then running:

```
apt-get update
```

in order to obtain the packages from the newly added repository. Nginx is then installed by running:

```
apt-get -t stretch install nginx-extras
```

or just

```
apt-get install nginx-extras
```

if you are running a newer version than Jessie.

After having installed Nginx, follow:

- <https://docs.docker.com/engine/installation/linux/debian/>
- <https://docs.docker.com/compose/install/>
- <https://github.com/jwilder/docker-gen>

to install docker, docker-compose and docker-gen.

Step 2 - Adding configuration files

Begin by running:

```
git clone https://github.com/microserv/deploy
```

to obtain the configuration files needed to deploy the system.

Now enter the newly created *deploy* folder, and replace all references to *despina.128.no* with your own domain name. This can also be done by executing the following command:

```
find . -type f -exec sed -i 's/despina.128.no/<YOUR DOMAIN HERE>/g' {} +
```

Then, copy the docker-gen configuration file *deploy/docker-gen/docker-gen.cfg* to */etc/* and the docker-gen systemd unit file *deploy/docker-gen/docker-gen.service* to */etc/systemd/system*.

After copying the files mentioned above, copy the nginx configuration file *deploy/nginx/nginx.conf* to */etc/nginx/* and the nginx docker-gen template *deploy/docker-gen/nginx.conf.tpl* to */etc/nginx/*

Now enable the docker-gen systemd unit file by running:

```
systemctl daemon-reload
systemctl start docker-gen
```

Step 3 - Launching the system

The system is launched using docker-compose. Docker-compose automatically fetches the Docker images required to launch the containers. The Docker images is by default fetched by Docker from a private repository hosted at 128.no:8080. To be able to access this repository, you must first log-in. To log-in execute:

```
docker login dockerisfun:QRN5A2So
```

Now enter the *deploy/docker* folder, and run:

```
docker-compose -d
```

to launch the system.

The system should be running and accessible through your domain after about 15 seconds. Then, access the administration panel by first accessing *YOUR-DOMAIN-HERE/auth/admin* and logging in using *admin* as the username, and *aStupidDefaultPasswordYouShouldChange* as the password.

After having logged in, go to *Applications*, then *Publishing*, and replace *despina.128.no* with your own domain in the *redirect URI* field. The system should now work as intended.

Kibana, elasticsearch and logstash, which is used to monitor the logs produced by the Docker containers can be launched by running the script *start-docker-logging.sh* in the *docker* folder. These services can be stopped by running *stop-docker-logging.sh* in the same folder. When these services are up, the logs can be read by accessing *YOURDOMAINNAMEHERE/kibana/*.

Appendix C

User manual

The core functionality of the system is to create and view articles. All main functionality is accessible through the main menu on top of the screen.

C.1 Creating an article

Step 1: In order to create an article, you will need to go to the editor page. Click the editor button as marked in C.1. If you are signed in and authorised, go to step 3, if not, continue to step 2.

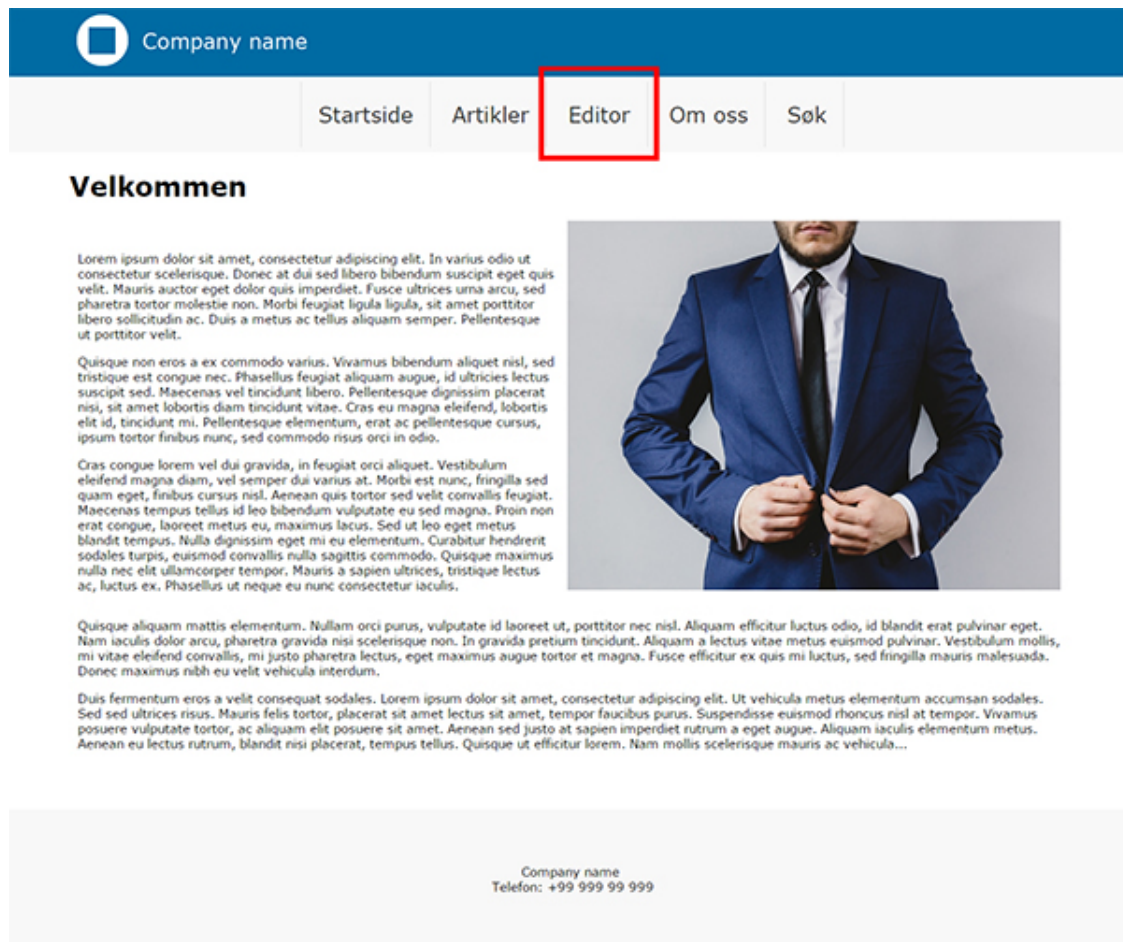
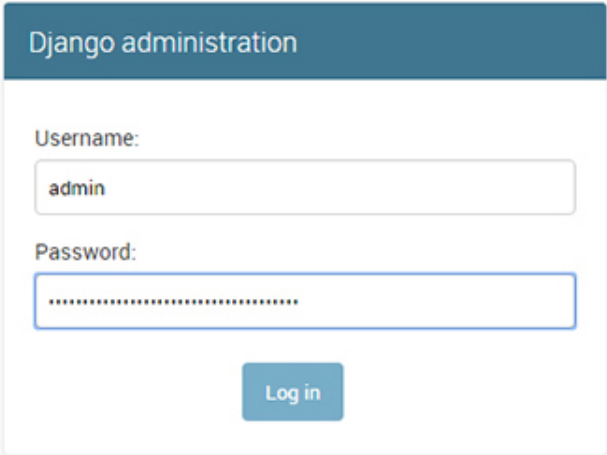


Figure C.1: Showing location of the editor button in main menu

Step 2: Fill in your credentials and log in as shown in figure C.2. This will prompt you with an authorisation screen. Click "Authorize" in order to get access (displayed in figure C.3).

A screenshot of the Django administration login interface. It features a dark blue header bar with the text "Django administration". Below the header, there are two input fields: "Username:" with the value "admin" and "Password:" with a masked password represented by dots. A blue "Log in" button is positioned below the password field. The entire form is centered on a light gray background.

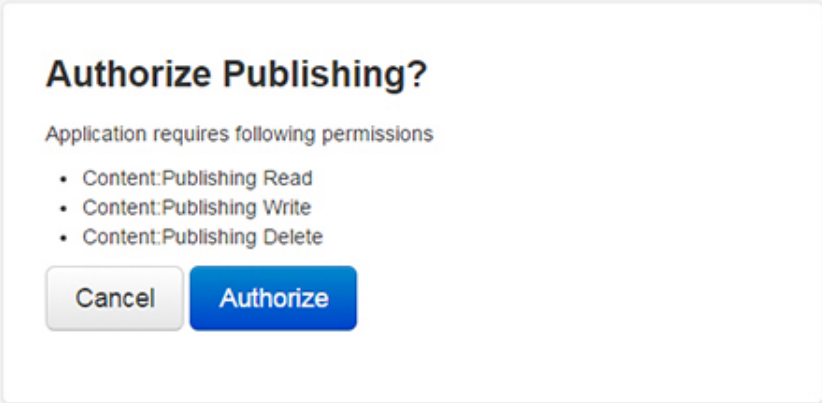
Django administration

Username:
admin

Password:
.....

Log in

Figure C.2: View for step 2

A screenshot of an "Authorize Publishing?" dialog box. The title is "Authorize Publishing?". Below the title, it says "Application requires following permissions". There is a bulleted list of permissions: "Content:Publishing Read", "Content:Publishing Write", and "Content:Publishing Delete". At the bottom, there are two buttons: a gray "Cancel" button and a blue "Authorize" button. The dialog box is centered on a light gray background.

Authorize Publishing?

Application requires following permissions

- Content:Publishing Read
- Content:Publishing Write
- Content:Publishing Delete

Cancel Authorize

Figure C.3: View for step 2

Step 3: You will then need to choose which template you want to base your article on from the drop down list titled "Velg en mal".

The screenshot shows the 'Artikkel' form in a web application. At the top is a blue header with a logo and 'Company name'. Below it is a navigation bar with links: 'Startside', 'Artikler', 'Editor', 'Om oss', and 'Søk'. The main form area is titled 'Artikkel'. A red box highlights the 'Velg en mal:' dropdown menu, which is open and shows three options: 'artikkel', 'nyhetsmelding', and 'tokolloneartikkel'. Below the dropdown is a rich text editor with a toolbar. The form includes fields for 'Tittel', 'Undertittel', 'Avisnitt', and 'Beskrivelse'. At the bottom right is a 'Submit' button. The footer contains 'Company name' and 'Telefon: +99 999 99 999'.

Figure C.4: Showing location of the template drop-down list

Step 4: Once you are done writing the article and adding metadata like title and description, you submit it by clicking the "submit"-button in the bottom-right corner.

The screenshot shows the 'Artikkel' form after some content has been added. The 'Velg en mal:' dropdown is now set to 'tokolloneartikkel'. The rich text editor contains the text 'Lorem ipsum' followed by two columns of placeholder text. The form fields for 'Tittel', 'Tags', and 'Beskrivelse' are filled with 'Lorem ipsum', 'dolor, sit, amet', and 'Generisk Lorem ipsum artikkel' respectively. A red box highlights the 'Submit' button in the bottom right corner. The footer remains the same with 'Company name' and 'Telefon: +99 999 99 999'.

Figure C.5: Showing location of the submit button

C.2 Searching

Articles can be found in two ways; either by finding them in the article list, as shown in figure C.6, or by searching. The following steps explain how to use the search function.

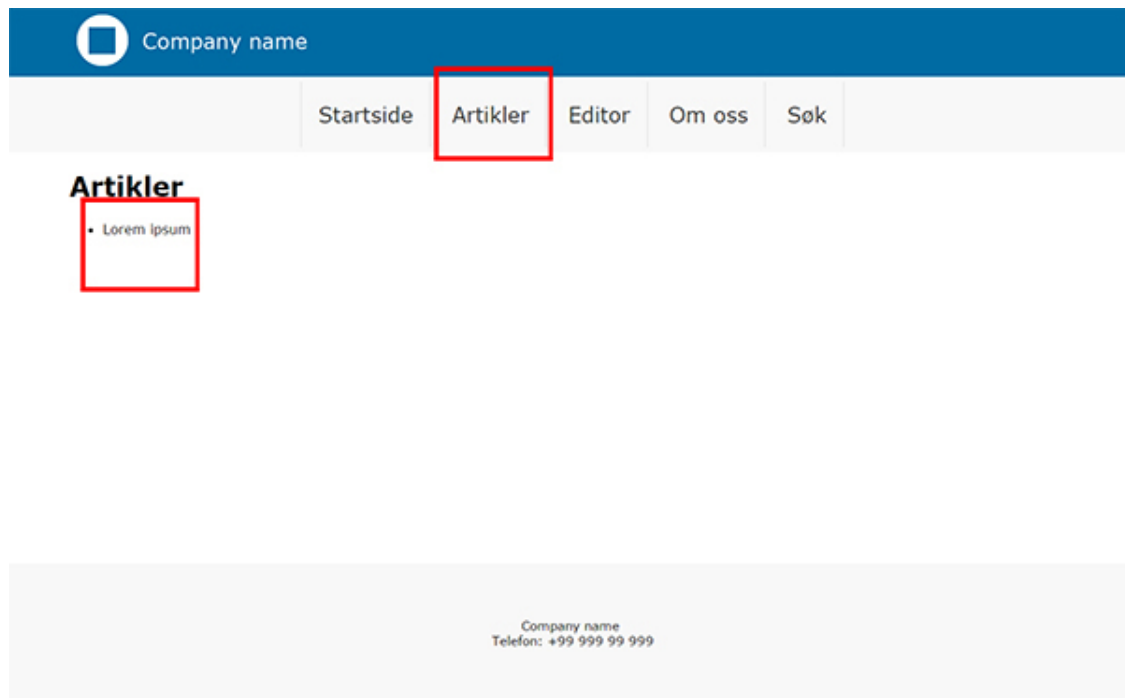


Figure C.6: Showing location of the submit button

Step 1: First, go to the search page as show in figure C.7.

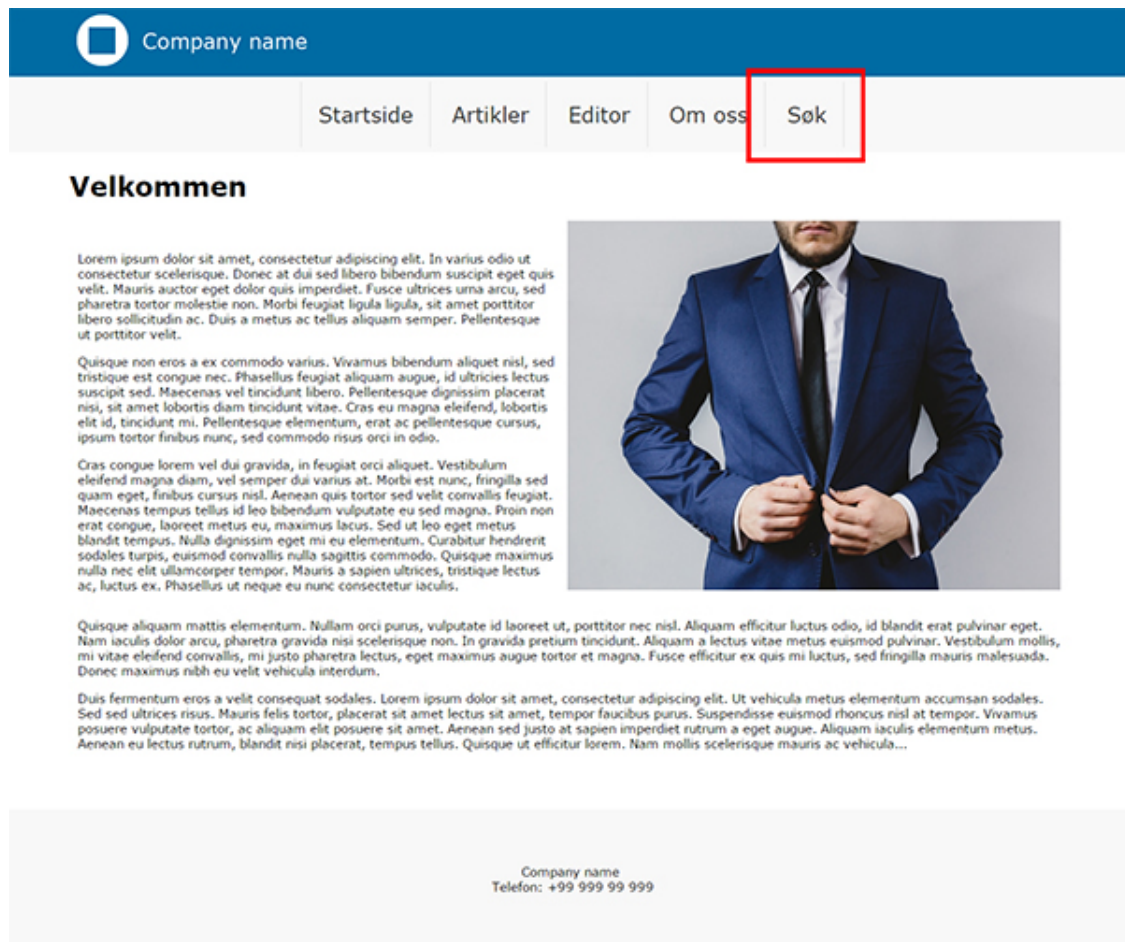


Figure C.7: Showing location of the search button in main menu

Step 2: When you start writing in the search field, it will give you suggestions of completions for potentially incomplete words. In order to accept this suggestion you can either left-click it or press the down-arrow on your keyboard.

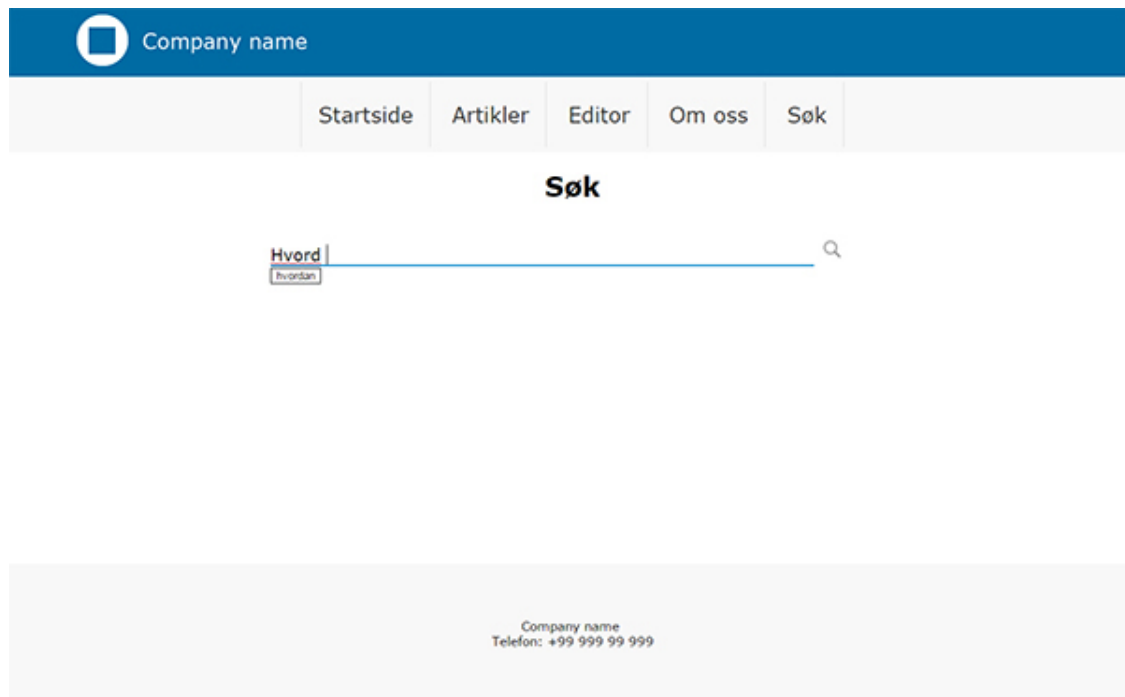


Figure C.8: Word completion suggestion

Step 3: Once your query is completed, you can search by either clicking the search icon or pressing the return key on your keyboard. This will present you with a suggestion to spelling in your query, which can be accepted the same way as suggestions in step 2. It will also present you with potential results, as shown in figure C.9.

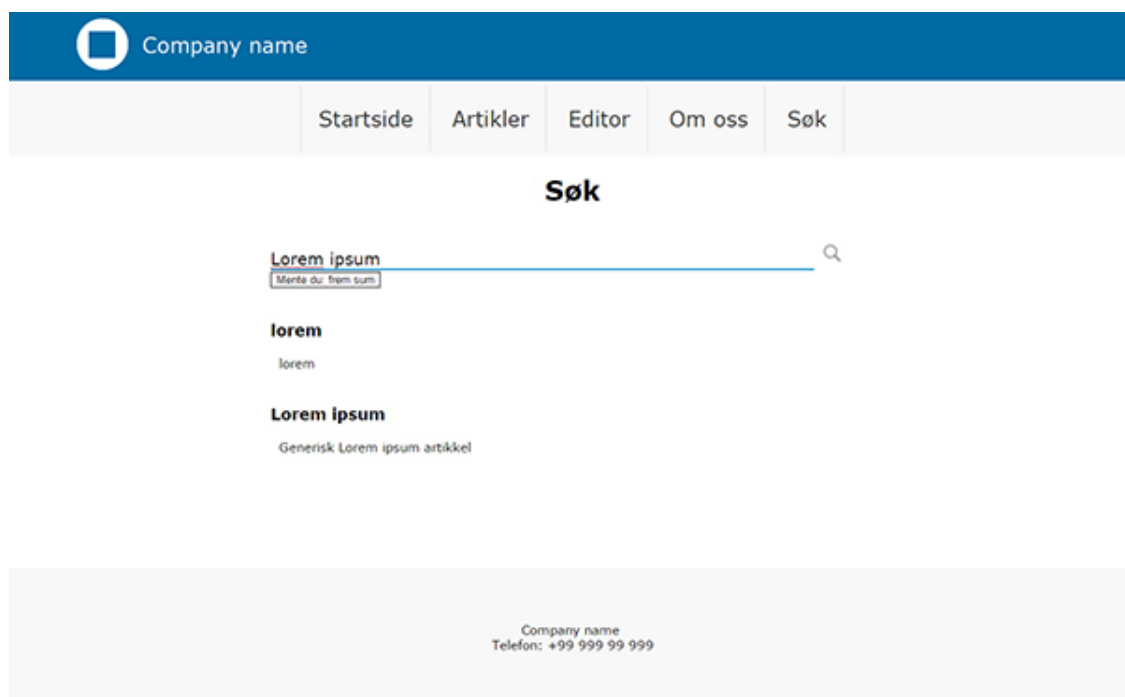


Figure C.9: Search results

Step 4: By clicking the article title you are brought to the full article as shown in figure C.10

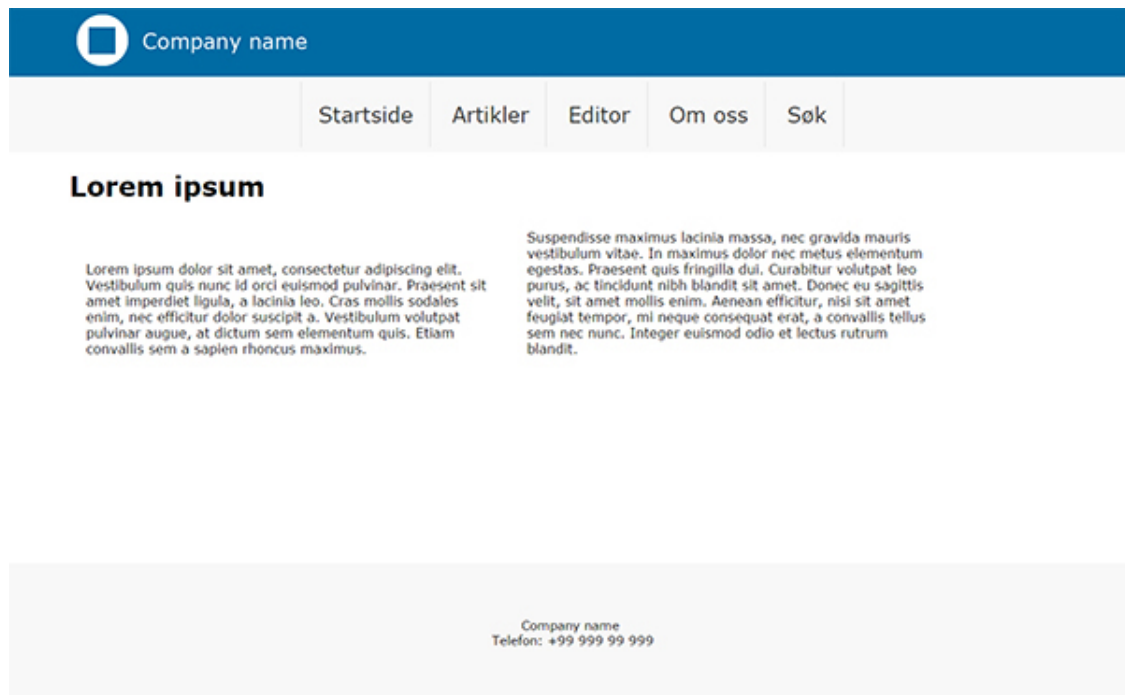


Figure C.10: Full article

C.3 Viewing service status

Step 1: Seeing as status is a developer feature, it is not clearly indicated with a button. In order to access it you need to click "Om oss" in the menu and then "Service status overview", as shown in figure C.11.

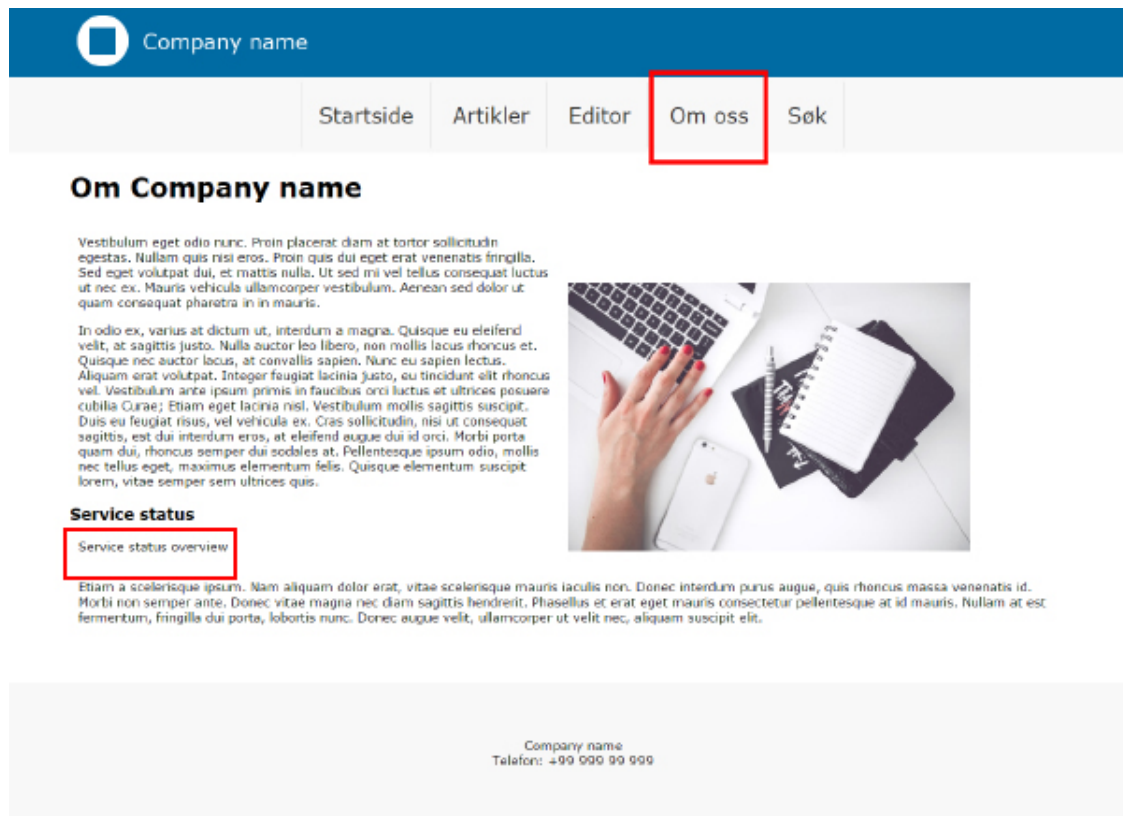


Figure C.11: Hidden status button

Step 2: By clicking 'show' on any of the services, you are provided extra information about its status. An example of the view for the indexer service can be seen in figure C.12.

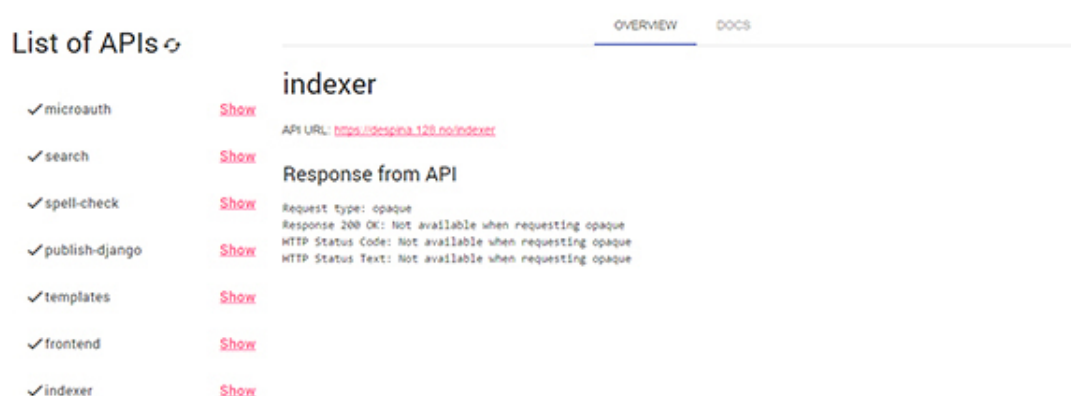


Figure C.12: Status overview. The symbol in front of the name represents its state. Check mark means available and an error symbol means it's unavailable.

Step 3: For viewing the documentation for the service, click on "Docs" in the upper-right corner as highlighted in figure C.13.

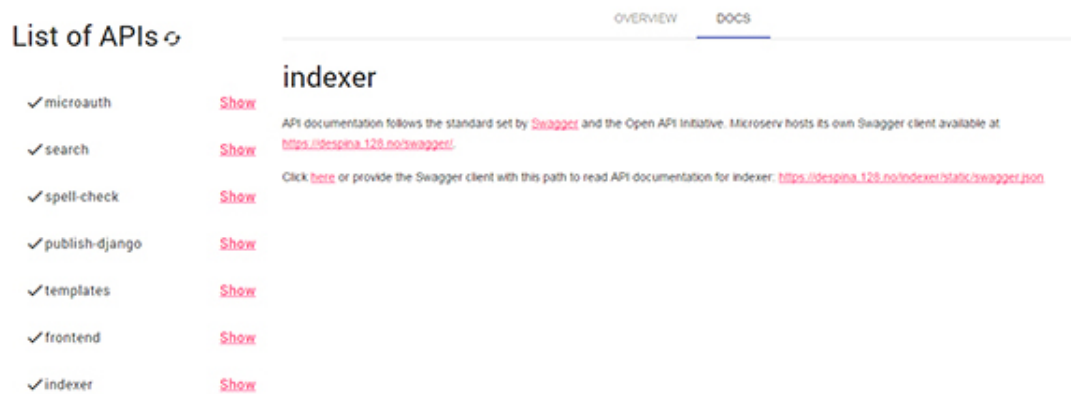


Figure C.13: Service documentation