IT2901 - Informatics Project II

GROUP 8 - STORYTELLING

Ragnhild Krogh

Espen Strømjodet

Kjersti Fagerholt

Eivind Halmøy Wolden

Hanne Marie Trelease

Audun A. Sæther

Roar Gjøvaag

2015-05-30 - Norwegian University of Science and Technology

dedication (optional)

Abstract

During this project a cross-platform application that provides personalized story recommendations was developed. These recommendations are developed by use of both content-based and collaborative filtering. In addition to accurate recommendations, an additional focus point was application usability. The developed application was named ”Vettu hva?”, and should hopefully encourage users to discover and read stories that they find interesting.

The first stage of development was spent refining requirements with the customer and to do a study of applications with similar functionality. Both of these elements were used to plan what the user interface would look like. The presented user interface is the result of several rounds of user testing and prototyping done to achieve a streamlined experience. The application and all its development stages are presented. The tools and technologies used are described with justifications for the usage of them.

The resulting application is able to recommend stories to a user in an esthetically pleasing manner. In addition to recommending stories, users can create and manage bookmarks as they see fit, and rate stories after reading them. The application achieved what it was meant to do, which was to serve as a tool for evaluating personalization techniques as a method to increase interest in cultural heritage. As the application was developed for a research trial, some of the non-functional requirements were relaxed since personalization functions were the main focus. We feel that the application meets its intended goal both functionally and esthetically.

Preface

This report describes a software development project completed by seven students in the IT2901 Informatics Project II course at Norwegian University of Science and Technology (NTNU), during the spring of 2015. The motivation behind this project was the assumption that personalization can lead to an increased interest in cultural heritage. The result of our work will be used to experiment with the personalization of cultural stories, to find out if this assumption holds true.

We would like to thank our customer at SINTEF: Jacqueline Floch and Shanshan Jiang. They have been dedicated to, and enthusiastic about, the project and provided lots of useful feedback to the group throughout the development process. The feedback did not only concern the development of the product, but also different versions of the report. For this, we are grateful. In addition, we would like to thank our main supervisor, Soudabeh Khodambashi, and the fill-in supervisor Nina M. Smørsgård for feedback on various versions of the report. This helped the group to think critically on how to achieve the best possible report.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This chapter introduces the customer, the team, and the project's definition and purpose.

1.1 Stakeholders

1.1.1 Customer

The employer for this research project was SINTEF, in this case represented by Jacqueline Floch and Shanshan Jiang. SINTEF is an independent multidisciplinary research organization within technology, science, social science and medicine. The organization has also provided assignments for the course IT2901 in the past.

1.1.2 Team

Table 1.1 lists the persons in charge of developing the project in this report, and some of their background competencies.

Table 1.1: Team description

Name	Competencies
Kjersti Fagerholt	HTML, CSS, JavaScript, Java, PHP, SQL, Python.
Roar Gjøvaag	HTML, CSS, JavaScript, Java, C#, Game Development, UX
Ragnhild Krogh	HTML, CSS, JavaScript, Java, Python, responsive web design
Espen Strømjordet	HTML, CSS, JavaScript, Java, UI Design experience
Audun A. Sæther	HTML, CSS, JavaScript, Java, PHP, SQL
Hanne Marie Trelease	HTML, CSS, JavaScript, Java, PHP, SQL
Eivind Halmøy Wolden	HTML, CSS, JavaScript, Java, PHP, SQL

As seen from **Table 1.1** the team had general experience with web design and computer application design. Some members had experience with making databases, resulting in not having to dedicate extra time for researching this topic. There was a mix of valuable experience between front-end and back-end development, as well as knowledge about

project management and how to relate to various actors such as users and other stakeholders.

1.2 Project description

In the course IT2901 [**?**] (Informatics Project II), at the Norwegian University of Science and Technology (NTNU), the main assignment was to develop a software project for a customer. This was done during the spring semester of 2015. The goal of the course was to gain practical experience with the development of a software process for a customer, covering the whole life-cycle of the software project.

The project described in this report is named Personalized storytelling. The purpose of this project was to create a cross-platform application (iOS and Android) which would allow users to discover personalized cultural and historical stories based on context-sensitive information and personal interests. The application is developed in connection to the EU IST research project *TAG CLOUD* [**?**], which aims at enriching cultural experiences through innovative mobile applications. In this project multiple ways of personalization had to be integrated to find good recommendations for the user. The project was originally two separate projects, a front-end project and a back-end project. The two was merged to one because there were not enough group members on them separately.

1.3 Problem description

Even though there is a rich cultural heritage in Norway, the public engagement in cultural heritage remains low. Museums and other cultural institutions have tried to increase interest with innovative exhibitions and tools. The motivation behind this project was the assumption that personalization can lead to an increased interest in cultural heritage. The resulting application will be used to experiment with the personalization of cultural stories, to find out if this assumption holds true. This was done by creating an application that presents personalized stories based on the user's interests and context, and thus encourages exploration and finding relevant and interesting stories to the user.

The application's name is "Vettu hva?", a name that was suggested by the customer and translates to "You know what?" in English. The name gives an impression of having something to tell, which is fitting for an app based on storytelling. The name also appeals to a

person's sense of curiosity, as if saying "I know something that you do not".

To summarize, this report details the entire development process of a mobile application, developed for Android and iOS. The application will provide its users with cultural stories in a personalized manner, with the goal of generating more interest in cultural heritage.

CHAPTER 2

REQUIREMENTS SPECIFICATION

This chapter describes the requirements for the application. The chapter is divided into two sections: one section describing the functional requirements and another that describes the non-functional requirements.

2.1 Functional requirements

The functional requirements were elicited and agreed upon with the customer in meetings and formalized in the requirement specification as seen in **Appendix C**. The first three requirements are non-functional requirements and therefore described in the next section. In early customer meetings the functionalities of the application were discussed informally. The group wrote a requirement specification which was then discussed at subsequent meetings. Each proposed requirement was refined and given a priority using a high-medium-low scale. This priority has guided the development process for the project. In addition, the group created its own prioritization where all the requirements were ranked, each with its own unique number ranging from one to the total number of requirements. The description of requirements in this chapter is more high-level than the one found in the requirement specification.

2.1.1 Summary of the functional requirements

- Sign up/Sign in view: The application should in some way be able to identify users, but keep them as anonymous as possible. As agreed with the customer, storing the email address is adequate, as it is only a prototype for a limited set of users and it does not store sensitive information. For research purposes, personal data like age group and gender should be collected.

- Preferences/Settings: The user should be able to specify some preferences regarding interest in cultural categories.

- Main view - Browse recommended stories: The application should provide the user

with recommended stories based on user preferences. The user should be provided with three choices for each story: to read it now, to reject the story, or to save it for later. In addition, the application should provide an explanation for why each story was recommended.

- Story view: The application should present a chosen story in a way that respects the work of the author and resembles the presentation given on *Digitalt fortalt*'s website. Every story should also include a link to the corresponding story on Digitalt fortalt. The user should be given the opportunity to rate the story and to bookmark the story.

- List view: The application should be able to keep lists of stories. Lists are created based on the bookmarks on stories, which are either system-generated (to-read, read) or the user's own bookmarks.

- Notifications: The application should provide the user with the opportunity to set a preferred time for receiving a notification about a new recommended story on their device. At the set time, the application should send the notification. In addition, the application should notify the user when a story is missing a rating.

- About site: The application should include an about site, which explains the context in which the application was created.

- Quick tour: An introduction to the application should be given to new users, and also be available through the menu.

- Personalization: When recommending stories, the application should employ both content-based filtering and collaborative filtering algorithms.

- Research: The application should gather information about usage. The customer provided a list of what this entailed, some details can be found in **Section 6.9**.

2.1.2 Use cases

The use cases in this section give an overview of the interaction with the system. In the requirements document one can find references to the use cases for requirements that involve external interaction.

A use case is a simple scenario that identifies actors involved in an interaction with a system and describe this interaction [**?** , p.106-107]. The use case diagrams here are presented with ellipses that represent the use case and the stick figures that represent the actors. Actors in these use cases are the user of the system, the device on which the application runs, and Digitalt fortalt. Include arrows means that the included use case is required to accomplish the use case the arrow originates from, while the extends arrow describe a use case that might be performed after the main use case. The use cases consist of a textual description as seen in **Tables 2.1 to 2.10** and corresponding use case diagrams as seen in **Figures 2.1 to 2.10**. The textual description follows a template which consists of these items:

- ID: A unique identification for the use case.
- Name: A short text describing the goal of the use case.
- Brief description: This is a more elaborate explanation of the use case than the above.
- Actors: These are the users/systems outside the application interacting with it.
- Priority: A metric describing the priority of this use case. The metric is derived from the functional requirements document and uses a high-medium-low scale.
- Preconditions: Describe what state the system should be in before the use case can start. Typically, some of the other use cases are already performed to set up the use case.
- Basic flow: This describes the normal flow from preconditions to postconditions in a numbered list.
- Alternate flow: A description of scenarios that differ from the basic flow described above. This includes exceptions and errors. It is also presented as a list, but the numbering in this list refers to the items in the basic flow list. The items in this list do not relate to each other.
- Postconditions: Describe what state the system should be in after the use case is performed.

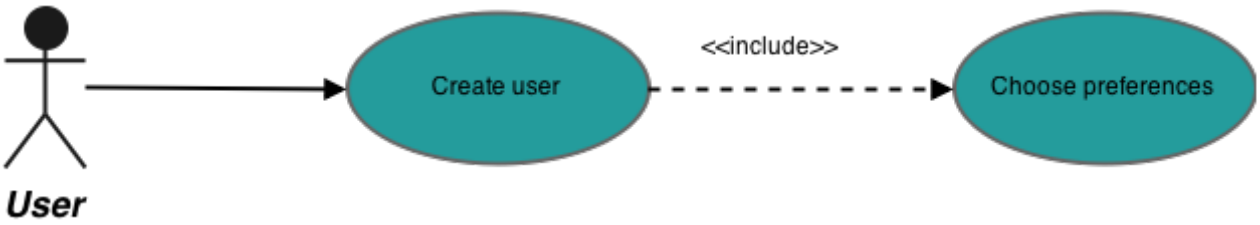


Figure 2.1: Use case diagram of U1. Create profile

Table 2.1: Textual description of U1. Create profile

ID	U1
Name	Create recoverable profile.
Brief description	Enter e-mail to register.
Actors	User
Priority	High
Preconditions	Application installed and no user is signed in
Basic flow	1. User clicks on register new user 2. User fills in e-mail into a registration form 3. Input validated 4. User finishes the registration and the system saves the user ID
Alternate flow	1. User skips registration and starts using the system 2. The system stores the new user by an ID internally
Postconditions	User is created and saved by the system

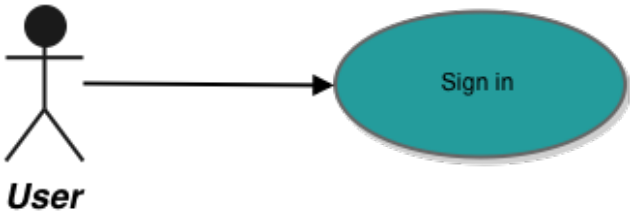


Figure 2.2: Use case diagram of U2. Sign in

Table 2.2: Textual description of U2. Sign in

ID	U2
Name	Sign in.
Brief description	Enter e-mail address to sign in.
Actors	User
Priority	High
Preconditions	User has already registered, but is not signed in on this device
Basic flow	1. User clicks on sign in 2. User fills in e-mail into form 3. System checks e-mail address. User id returned 4. Main view displayed
Alternate flow	1. User not found 2. Error message displayed, go to 2.
Postconditions	User is signed in to the system

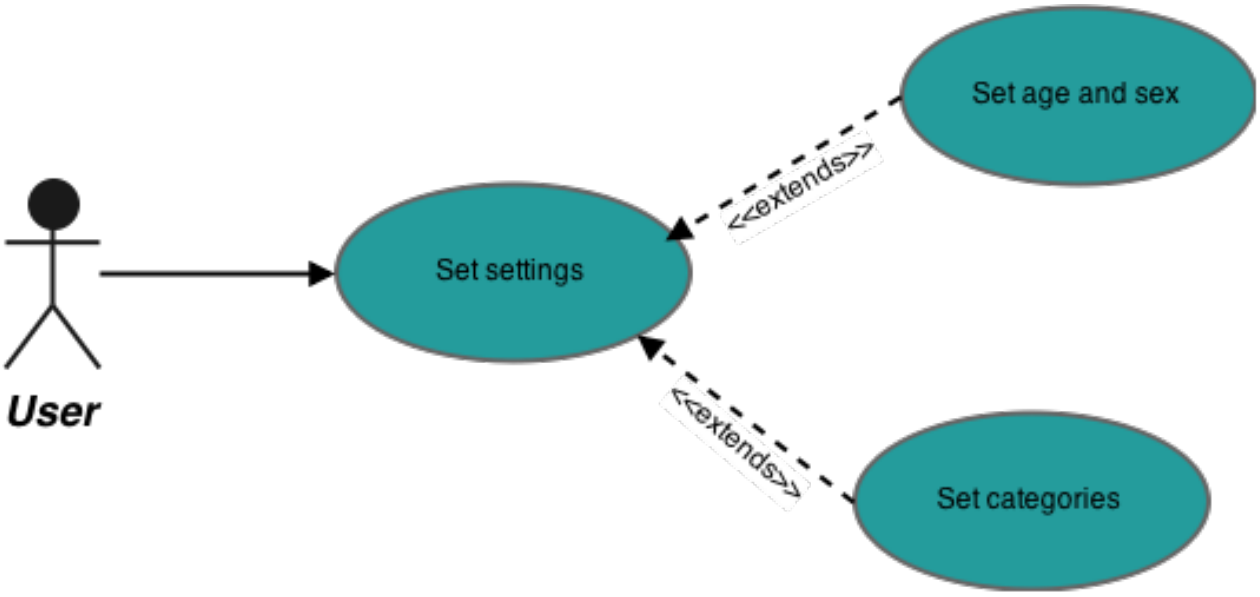


Figure 2.3: Use case diagram of U3. Set initial settings

Table 2.3: Textual description of U3. Set initial settings

ID	U3
Name	Set initial settings.
Brief description	Specify user information to be used for receiving stories.
Actors	User
Priority	High
Preconditions	Application installed, user id received
Basic flow	1. User is initially asked to fill a form about preferences by the system (a) Personal info: age group and sex (b) Cultural category preferences 2. The system saves information about the user's preferences
Alternate flow	1. User clicks on settings in order to change preferences
Postconditions	The system has information about the user in order to provide personalized story recommendations.

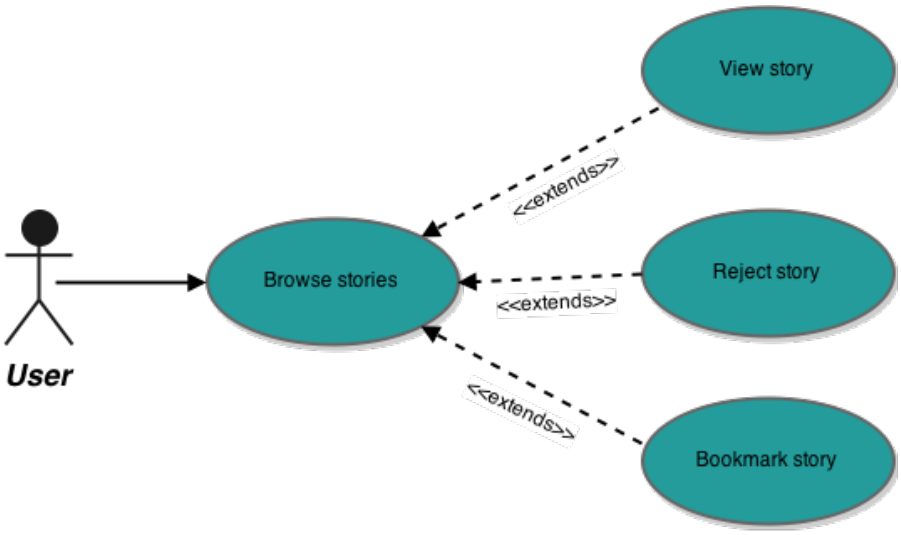


Figure 2.4: Use case diagram of U4. Browse recommended stories

Table 2.4: Textual description of U4. Browse recommended stories

ID	U4
Name	Browse recommended stories.
Brief description	User is shown a list of recommended stories to choose from.
Actors	User
Priority	High
Preconditions	Preferences already set
Basic flow	1. User is shown recommended stories including an explanation of why the story was recommended 2. User does one of the three following actions on a story: (a) Choose to read the story now (b) Reject the story (c) Bookmark story
Alternate flow	
Postconditions	The system has stored information about the choice of the user.

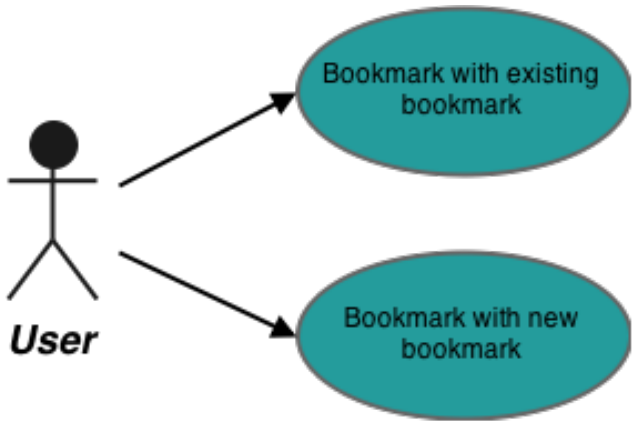


Figure 2.5: Use case diagram of U5. Bookmark story

Table 2.5: Textual description of U5. Bookmark story

ID	U5
Name	Bookmark story.
Brief description	Bookmark story as to-read or use user-defined bookmarks.
Actors	User
Priority	Medium
Preconditions	User is registered and signed in
Basic flow	1. User opens a story to read 2. User clicks the bookmark button in the story and selects the desired bookmark to add to the story. 3. Story will be bookmarked with the chosen bookmark.
Alternate flow	1. The desired bookmark does not exist. The user clicks to add new bookmark.
Postconditions	Bookmarks will be added to and/or removed from the story according to user's actions

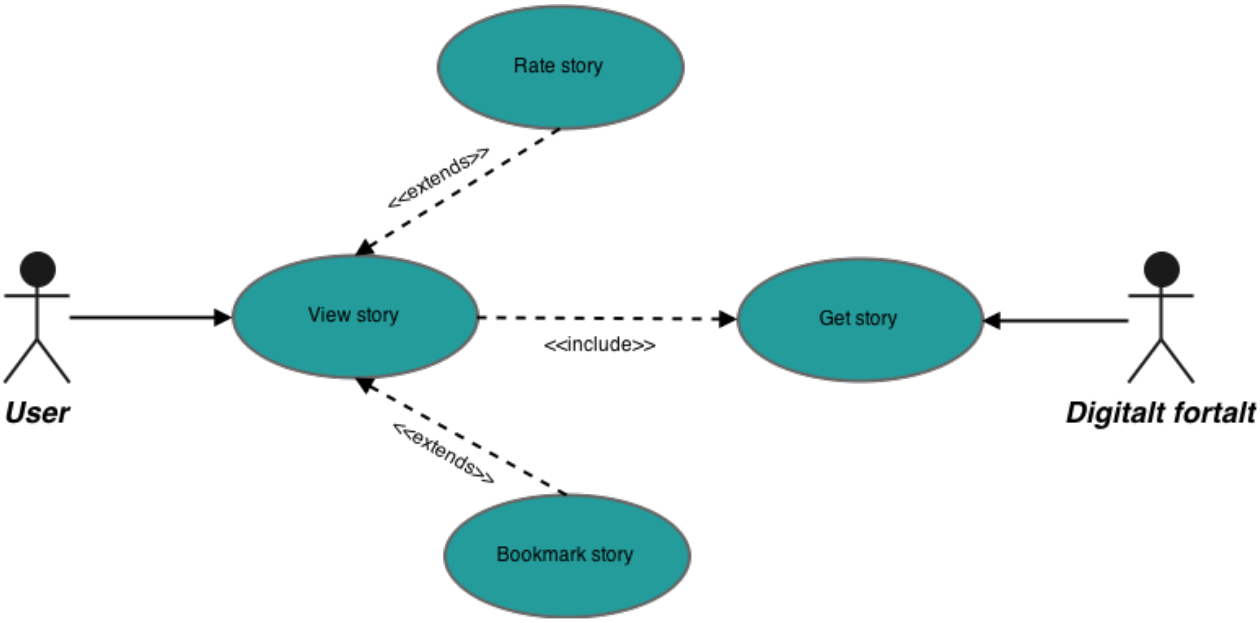


Figure 2.6: Use case diagram of U6. View story

Table 2.6: Textual description of U6. View story

ID	U6
Name	View story.
Brief description	Display the selected story.
Actors	User, Digitalt fortalt
Priority	High
Preconditions	The story has been recommended to the user at some point and is either in the current recommendations list or is bookmarked by the user.
Basic flow	1. User selects story 2. Display: (a) The story (b) Available media formats (c) Link to story on Digitalt fortalt 3. User may select a media format
Alternate flow	
Postconditions	The story is shown according to the user preferences

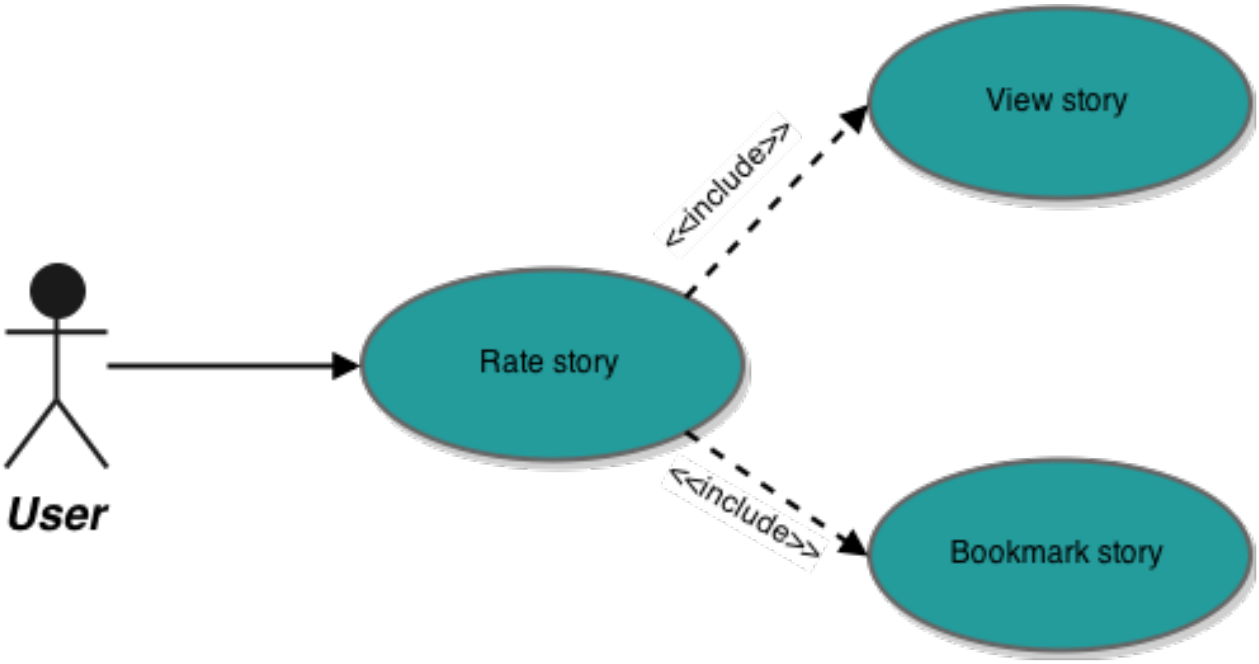


Figure 2.7: Use case diagram of U7. Rate story

Table 2.7: Textual description of U7. Rate story

ID	U7
Name	Give feedback / rating.
Brief description	Give a rating on a story.
Actors	User
Priority	High
Preconditions	User has opened a story for reading
Basic flow	1. After reading the story, the user clicks on a rating to give feedback on the story 2. The system saves the rating 3. Story will be bookmarked with ”Lest”
Alternate flow	1. User does not give a rating on the story before closing it 2. The system reminds the user to rate the story at a later time
Postconditions	The rating of the story from the user is saved

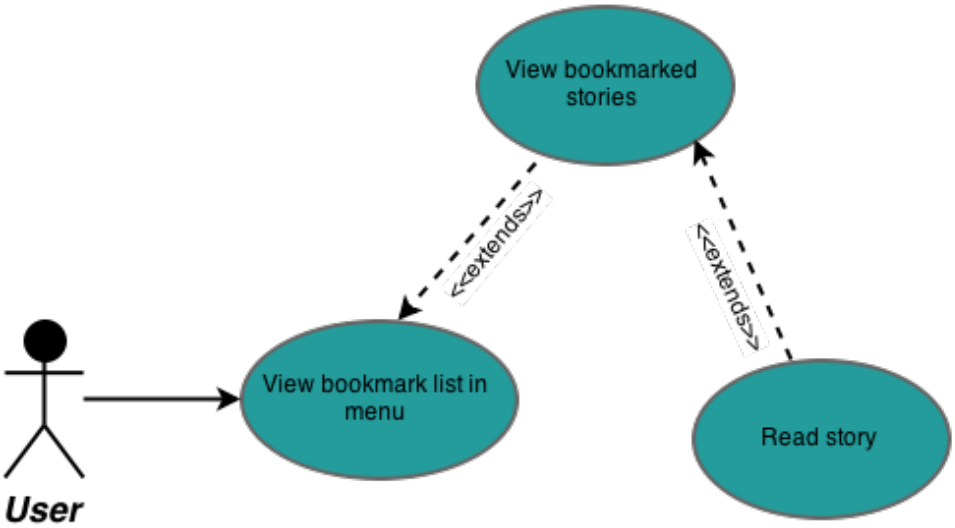


Figure 2.8: Use case diagram of U8. View bookmarked stories

Table 2.8: Textual description of U8. View bookmarked stories

ID	U8
Name	View bookmarked stories.
Brief description	View stories connected to a bookmark (read,to-read,bookmarks defined by user).
Actors	User
Priority	Medium
Preconditions	User is registered and signed in
Basic flow	1. User selects bookmark from menu 2. List of stories shown
Alternate flow	1. No stories found 2. Display message “No stories found”
Postconditions	User is shown all of the stories collected

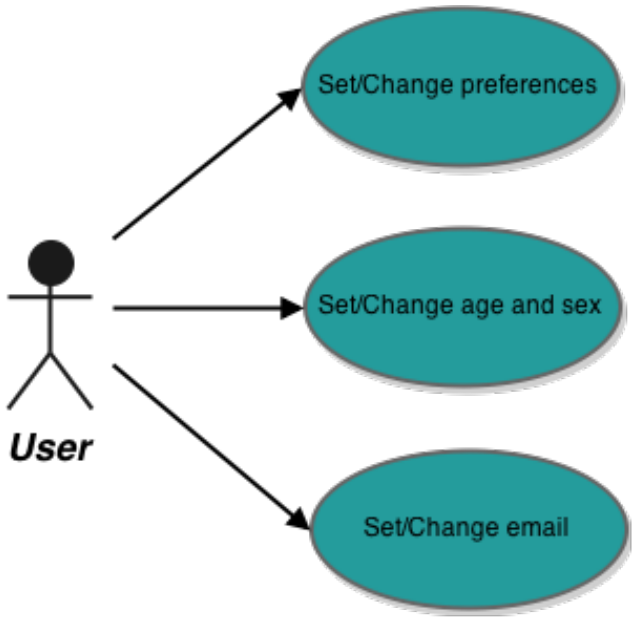


Figure 2.9: Use case diagram of U9. Specify settings

Table 2.9: Textual description of U9. Specify settings

ID	U9
Name	Specify settings.
Brief description	Specify and change system settings.
Actors	User
Priority	High
Preconditions	User is signed in
Basic flow	1. While logged in, user clicks on settings 2. User edits the data in settings (a) Category preferences (b) Age group / sex (c) User e-mail 3. The system saves the new settings
Alternate flow	
Postconditions	Settings are changed and saved

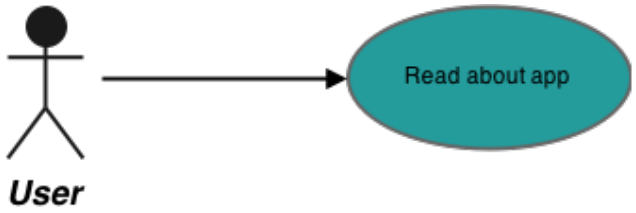


Figure 2.10: Use case diagram of U10. Read about app

Table 2.10: Textual description of U10. Read about app

ID	U10
Name	Read about app.
Brief description	Basic info about the application.
Actors	User
Priority	High
Preconditions	User is signed in
Basic flow	1. While signed in, user clicks on settings 2. User clicks on read about app and is presented with information about the app
Alternate flow	
Postconditions	

2.2 Non-functional requirements

A general requirement for the project was to use English in all parts related to the documentation of the application, while the language in the application would be Norwegian. Other general requirements concerned the platforms the application should run on. It was decided that it should run on both Android and iOS, and that the design of the application should approach a native feel as much as possible on these platforms.

A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of the stakeholders. You can think of a quality attribute as measuring the product along some dimension of interest to a stakeholder [**?** , p.63]. To make better decisions at a top-level design perspective, and to make better decisions on a component and implementation level, the team wanted the customer to rank each of the quality attributes. The list of QAs and the ranking can be seen in **Table 2.11**.

Table 2.11: The quality attributes used in this project. Prioritized by the customer

Quality attribute	**Description**	**Priority**
Usability	How easy is it for the user to accomplish a desired task and what kind of user support should the system focus on. (e.g. tutorials or hints)	1
Monitorability	How important is the ability to monitor how the system performs while it is executing. (e.g. statistics)	2
Modifiability	How important and how easily should the product be able to be changed after it is finalized? (e.g. making changes to the UI)	3
Performance	How important are the timing and speed of the system? (e.g. response time for fetching stories)	4
Interoperability	How important is the ability for the system to work together with other systems? (e.g. making use of specific communication protocols or the use of a specified data format)	5
Availability	How important is the reliability of the product? The easy way to estimate this is uptime of the service.	6
Security	How important is the system's ability to protect data and information from unauthorized access. (e.g. losing personalized data)	7
Testability	How important is the ability to set up tests for the system (e.g. setting up automated test for components and parts of the system)	8

The described quality attributes were relaxed requirements for the application, as the per-

sonalization aspect was considered most important by the customer. The reader should also keep in mind that there are crossover attributes, e.g. Maintainability/Testability may increase availability, which could be included in a project of greater scope.

The ranked list was helpful in choosing the solutions, architecture and patterns that were most in accordance with the customer's vision and needs. In a project with more resources or a more narrow scope there would have been more plans for measuring and quality assure each of the attributes, but this would broaden the scope and workload of a project which was already on a tight schedule. Instead this was used as a guideline assisting the development and decision making throughout the project. This was not a decision that was easily compromised but was believed to be one area we could free up some time. With that said, extensive testing and measuring was done to achieve a high quality product (see **Chapter 8**).

CHAPTER 3

PRE-STUDY

This chapter discusses the research that had to be done, and the choices that the team made in relation to choosing development frameworks and technologies. The chosen development process for the project can be found here. The chapter also describes some already existing applications that are similar to this one, and a study about personalization which was a key aspect of this project.

3.1 Project assumptions and constraints

In this section, assumptions and known constraints for the project are addressed. The list of assumptions are details assumed ahead of the documented project requirements, while the constraints specifies matters which would impact the project.

Assumptions on which the planning of the project was based:

- Access to an Application Programming Interface (API) for Digitalt fortalt would be provided by the customer.
- The customer would give access to a server to deploy the back-end part of the application on.
- The frameworks chosen for development had the features needed.
- The customer would be available for weekly meetings.
- Each member of the team would be able to work about 20 hours a week.
- The team would meet at least three times a week.
- The team would follow the set ground rules as seen in **Appendix A**.

Constraints which the team had to work within throughout the project:

- The deadline for delivering the project was May 30th, and there was no possibility to extend it.

- The front-end should be developed for both the iOS and Android platforms.
- The team consisted of 7 people, with little previous experience in mobile application development and no knowledge of personalization algorithms.
- The application is under the Apache License version 2.0 [**?**]. In summary, this grants copyright and patent rights for users to further distribute it, or distribute a modified version using the same license. It should be clear what the eventual modifications are. The source code can also be used as a part of a closed source project.
- The customer would cover the cost of software tools if necessary, but encouraged the team to use freely available tools if possible.

3.2 Choice of framework

As one of the requirements from the customer was that the application should be cross-platform (Android and iOS), a hybrid app was found to be the best option. The alternative would have been to create two separate native apps for both platforms. However, this would have required more work and complicated a delivery within the deadline, especially as the team had little experience with developing for either platform. A hybrid app is a web app made with HTML and JavaScript wrapped in a native shell so that it can be run like a normal native app. This was a big advantage, as the team already had experience with web design. It also makes it possible to make use of the many tools available for web development, and most of the code can be reused for multiple platforms. Bad performance used to be a disadvantage to the hybrid approach, however mobile hardware has improved significantly in the last few years, so this is not a major issue today. The following sections will discuss the advantages and disadvantages of different frameworks that were considered, and explain the final choice for this project.

Table 3.1 summarizes some of the capabilities and limitations of the various frameworks that were under consideration.

3.2.1 PhoneGap

PhoneGap is an open-source mobile app framework for native packaging [**?**]. What it does is take in a mobile app consisting of HTML, CSS and JavaScript files, and wrap it in a native shell. It can then be deployed to iOS, Android and Windows 8. It also gives

Table 3.1: Framework comparison

	PhoneGap w/ Ionic	**Appcelerator Titanium**	**PhoneGap w/ Sencha Touch**
Can write a single code that runs on both iOS and Android?	Yes	No	Yes
Access to native components	Yes	Yes	Yes
Expected ease of learning and use	Relatively easy	Difficult	Somewhat difficult
Performance of created application	Good, AngularJS also provides a performance boost	Very good, as it provides direct access to native components	Acceptable, but not as performance- focused as Ionic.
Debugging	Easy	Difficult	Easy
Programming language used	HTML5, CSS, JavaScript with AngularJS	JavaScript	HTML5, CSS, JavaScript

easy access to the native features of the phones (geolocation, notifications, storage, etc.) through different APIs. It is not necessary to think about the native software development kits (SDKs), as the app will be compiled and built with the newest SDK for the platform. It is a very popular framework, and there are many plugins created for it that provide additional functionality.

3.2.2 Ionic

Ionic is an open-source UI framework focused on making it easier to create hybrid mobile apps with a native feel [**?**]. It accomplishes this by offering a foundation to build on and UI components based on design patterns and best practices found in native apps. The foundation can be built on and customized with additional HTML, CSS and JavaScript. Part of the foundation is AngularJS, which is a JavaScript framework that extends HTML to make dynamic views in web applications. It gives the app a modular architecture, which means that code can easily be reused for both iOS and Android. A disadvantage is that it is necessary to take time to learn AngularJS in order to take full advantage of Ionic. As Ionic is one of the most popular hybrid mobile frameworks, there exists more learning material about it, and it also has an active user community. Performance is not quite as good on older devices, especially if using large amounts of animations or media.

3.2.3 Appcelerator Titanium

Titanium is a cross-platform Javascript runtime and API framework [**?**]. It currently supports iOS and Android. It offers a JavaScript API which gives access to native UI components and features for the specific platforms, instead of trying to replicate it with CSS or JavaScript like other frameworks. This gives the app a performance advantage, and it is easier to make the interface and interactions feel native. Because the APIs are platform-specific you have to write separate versions of your app for the different platforms. Another disadvantage is that it is difficult to debug as there is no good debugger and Titanium projects cannot be run in Xcode. There would also have been more to learn to be able to use it, as it does not use HTML/CSS.

The customer mentioned this framework as a potential choice, because a previous group of students used this in a related project stedr (see **Section 3.6.1**). The team got into contact with one of the developers of stedr, who expressed that the framework worked well, but that it took a while to learn and there was difficulty with integrating the app on iOS.

3.2.4 Sencha Touch

Sencha Touch is a mobile framework with a large number of UI components and an architecture for the front-end [**?**]. Instead of enhancing an HTML file, it generates the document object model (DOM) with JavaScript. The DOM is an interface which creates a representation of the HTML code and allows the developer to manipulate the HTML elements. Sencha Touch can be used with PhoneGap, but also has its own native packager. It supports the platforms iOS, Android, BlackBerry, and Windows Phone.

3.2.5 Conclusion

The framework combination chosen for this project was Ionic and PhoneGap, as this seemed to fit this project's properties and requirements the best. They are both some of the most popular hybrid frameworks and they are a common combination to use. A prototype can quickly be set up with Ionic, and then iteratively customize it to fit the requirements and create a good user experience. One of the non-functional requirements was that it should be easy to extend the application and to reuse parts of it for other apps. This is fulfilled by Ionic's modular structure. PhoneGap makes it easier to wrap the code

in a native iOS and Android shell. This process could also have been done manually, but it requires some knowledge about the native code languages and SDK. The many plugins available for PhoneGap should also cover the need for use of native features.

3.3 Software development process

The choice of which development process to use in the project was a central decision to make. The following sections describe the different models that were under consideration by the team, as well as some advantages and disadvantages of each, which influenced the decision of which one to be used for the project. **Table 3.2** below gives a comparison of some aspects of the various processes that were considered relevant for the project.

Table 3.2: Development process comparison

	Scrum	**Extreme Programming**	**Waterfall**
Type of methodology	Agile	Agile	Plan-driven
Responds well to changes in requirements?	Yes	Yes	No
Amount of planning needed	Moderate	Very little	Very much
Level of detail for project plan	Moderate	Low	High
Customer involvement	High	High	Low
Frequency of testing	Continuously	Continuously	Only near the end
Iteration length	Short	Short	Long

3.3.1 Waterfall model

The waterfall model is a plan-driven process with well-defined phases. These phases normally include requirement analysis, system design, implementation, testing, and maintenance [**?** , p.30-32]. It is necessary to finish one phase before starting the next, and due to this, most planning and decisions need to be made at an early stage in the development. As such it is difficult to respond to changes in the requirements. Another issue with this model is that iterations often involve a significant amount of rework and it is normal to postpone some parts of the iterations in order to continue with the later stages of development. This can lead to errors in the system as well as bad design choices.

3.3.2 Extreme programming

Extreme programming is an agile method focused on pushing out new system versions and functionalities rapidly [**?** , p.64-72]. All requirements are written as user scenarios,

and before writing the code it is necessary to develop tests for the task. Team members program in pairs and when the code passes all the tests, it can be integrated into the system. It is common for a customer representative to take part in the development and make acceptance tests. New system releases are regularly presented to the customer, and this way it becomes easier to cope with changing requirements. Some of the drawbacks of extreme programming include the lack of overall plans for the project. Several documents such as design details and overall report are left out, and it lacks a solid plan for when to implement the various functionalities.

3.3.3 Scrum

Scrum is a general agile method with focus on managing iterative development rather than specific technical engineering approaches [**?** , p.72-74]. It also allows for a rapidly changing development environment and close collaboration between the members of a team. To facilitate this, scrum makes use of phases called sprints, daily scrum meetings, and several types of charts and logs. One of the main challenges for a scrum team is to choose the right amount of work per sprint so that they do not end up with too little or too much work. Scrum has several similarities with extreme programming, such as high involvement of the customer in the development, as well as continuous testing while implementing new functionality.

3.3.4 Conclusion

For this project, the agile software development methodology scrum was used. The customer wanted to keep the flexibility to adapt the requirements based on the feedback from the users involved in the usability tests. Scrum would be helpful in this regard as it would allow the group to quickly make a simple application which could be tested by the users and customer. Also, an agile process was beneficial as it allowed the team to be flexible and rapidly respond to changes in requirements.

3.4 Personalization

To provide story recommendations in accordance with each user's interests the content needs to be personalized. Personalization involves using technology to tailor content, to individual users' characteristics or preferences, and to accommodate the differences between individuals. It is a way of meeting the user's needs by making interactions faster

and easier, which will hopefully increase user satisfaction and the likelihood of repeat visits. Personalization may be achieved using recommender systems.

3.4.1 Recommender systems

Recommender systems are software tools and techniques that attempt to provide recommendations of items [**?**]. Such systems are simply information filtering systems with the goal of providing suggestions for items to be of use or interest to a user. A few examples of items used in this context are movies, music, books and products in general. Recommender systems typically produce a list of recommendations. The two most common approaches to produce such a list are content-based filtering and collaborative filtering.

3.4.2 Content-based filtering

Content-based filtering methods are used to find similarities between a user's preferences and the description of an item [**?**]. These algorithms try to recommend items that are similar to items a user has liked in the past or is looking at in the present. Items that a user likes or has interacted with can be seen as a part of the user's profile. Content-based filtering depends on there being much descriptive data available on the items. To find items to recommend, items are compared against a user's profile, and recommendations are given based on how well they match the profile. User feedback, usually in the form of rating or a like or dislike button, can be used to assign weights to certain attributes. By using user feedback and weighting it is possible to give more accurate recommendations. [**?**]

3.4.3 Collaborative filtering

Collaborative filtering is based on collecting and comparing information on users' behavior, activities or preferences and to recommend items based on a user's similarity to other users [**?**]. This approach tries to predict what a user will like based on what similar users have liked. Collaborative filtering assumes that users who have agreed in the past will agree in the future, and that they will like similar items as they liked in the past. These methods often suffer from two problems; cold start and sparsity. Collaborative filtering often requires a large amount of existing data on users to be able to make accurate recommendations. The cold start problem is the absence of such data at the beginning of a project. The sparsity problem is that collaborative systems are dependent on having

many active users to properly distribute ratings across all the items in the system. However, most active users have only rated a few items in the overall database, which means that even the most popular items could have very few ratings. The greatest strength of these techniques is that they are independent of any documented representation, e.g. textual descriptions and subject-tags, of the objects being recommended and work well for objects that are difficult to define such as music and movies.[**?**]

3.5 Open source recommendation tools

To make the application's recommender engine an open source recommendation tool was chosen. The team recognized that it would be a consuming task to write the personalization code ourselves. By employing an open source tool the recommendations would most likely be more correct, seeing that the open source alternatives have been more thoroughly tested and researched than this team would accomplish with such a short timeframe. The customer provided a few recommender alternatives, which is described in this section. **Table 3.3** display the different tools' language use, filtering type used and amount of documentation.

Table 3.3: Recommendation tool comparison

	Mahout	**LensKit**	**Duine**
Language	Java	Java	Java
Filtering type	Collaborative, Content-based possible	Collaborative, Content-based possible	Hybrid
Documentation	High	High	High

3.5.1 Apache Mahout

Apache Mahout is an Apache Software Foundation project with the goal of building an environment for quickly creating scalable machine learning applications [**?**]. The algorithms primarily focus on the areas of collaborative-filtering, clustering and classification. Mahout mainly focus on collaborative filtering, however it is possible to implement content-based filtering using Mahout's item-based collaborative filtering. Mahout uses Java as the main programming language, and implements both user-based and item-based collaborative filtering.

3.5.2 LensKit

LensKit [**?**] is a Java-based recommender toolkit developed by GroupLens [**?**]. It provides an API for recommendation algorithms, an evaluation framework for evaluating recommender performance and implementations of common collaborative filtering algorithms. It is intended to be useful for research and educational purposes. The aim of the project is to produce best practice implementations of common algorithms. Like Mahout, LensKit mainly focus on collaborative filtering, but it is possible to implement content-based filtering using LensKit's item-based collaborative filtering. LensKit implements user-based and item-based collaborative filtering.

3.5.3 Duine

Duine framework has been developed by the Telematica Instituut/Novay and is based on personalization research, specifically into recommender systems [**?**]. It is a software library for prediction-engine development, which uses Java to implement collaborative filtering techniques in addition to information filtering (hybrid recommendation). It dynamically switches between each prediction given the current state of the data. For example if there are few ratings available, it uses the content-based approach, and switches to the collaborative when the scenario changes.

3.5.4 Conclusion

It was difficult to find much information about the different open source alternatives, other than goals and algorithms used. There was no definite performance comparison of the different recommenders, so it was hard to know which of them was best. However, during the research, Mahout was the recommender which was most often referenced in articles and forums. These sources gave the impression that Mahout was easy to use as a "first-time" recommendation engine, and that the recommendations were close to accurate. It was also desirable that Mahout could provide both collaborative and content-based filtering. As a result Mahout was the chosen recommender.

3.6 Existing solutions

Our application is not the only application that has been developed in connection to the TAG CLOUD project. This section evaluates two of these applications, namely *stedr* and

Table 3.4: Summary of the main findings in the evaluation of existing solutions

	stedr	**Cooltura**	**Magic Tate Ball**
Content	Stories from Trondheim and other cities in Europe	Stories from three pilot sites, including Trondheim	Artworks and some information about them
Usability	- Requires knowledge of the location to find sites on map - The picture occupies much of the space in the site and story view - Informative and useful help site - Clear and consistent language	- List of locations is easier to navigate than a map and does not require knowledge of the exact location of the sites - Picture is dynamic and makes room for the relevant information to the user when scrolling - Lacks a help site	- Two options for starting the application and for browsing the artworks accommodates different user groups - Visibility of system status when processing
Personalization	Does not provide any personalization feature	Not implemented in the tested version	Provide personalization using different input parameters.

Cooltura. Both of these applications present stories regarding cultural heritage. Since this project included personalization, an evaluation was also performed on the application *Magic Tate Ball*. This application was chosen because the customer mentioned this as a possible inspiration for the current project.

These three applications were evaluated using the following criteria:

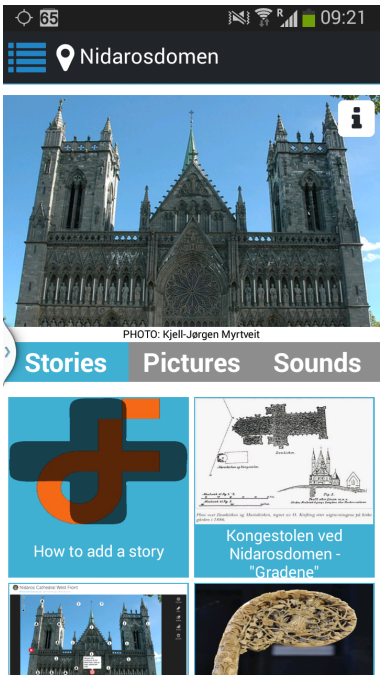
- Content. Does the application provide satisfying content or is something lacking?
- Usability. Usability concerns how easy it is for the user to accomplish a task. (see **Section 2.2** for a more elaborate definition). The evaluation here draws on Jacob Nielsen's ten usability heuristics as defined in [**?**].
- Personalization. To what extent does the application provide the user with the opportunity for individualized content?

The main findings in the evaluation are summarized in **Table 3.4**.

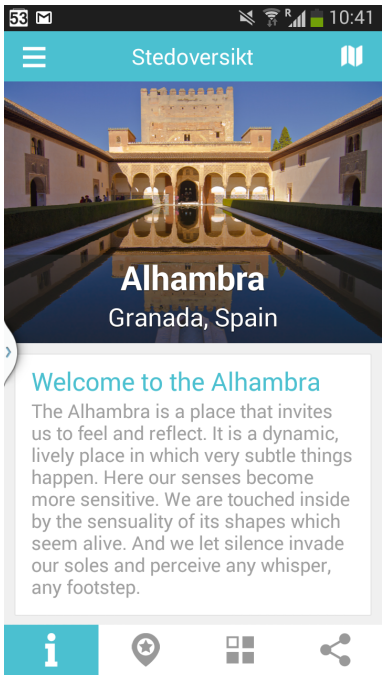
3.6.1 stedr

The stedr application was developed by students as a prototype to test which forms of storytelling can contribute to engaging people in cultural heritage. It also allows users to create their own stories, to evaluate what kind of cultural stories amateurs are willing to produce. Content was not the primary focus in the application, however it contains stories related to many cities in Europe. A possible problem in the application is finding the site the user is looking for, since the main view consists of a map with nondescript markers representing each available site.

The site view in stedr can be seen in **Figure 3.1a**. To see all the content, one might have to scroll down. The main picture is static when scrolling, occupying the upper half of the screen, which is also the case when viewing a specific story in the story view. It might be argued that Nielsen's heuristic of aesthetic and minimalist design in case of scrolling is violated as the picture's importance in the dialogue is less important than the space it occupies suggests.



(a) A screenshot from stedr showing the available options for one site



(b) A screenshot of Cooltura showing the view for one particular site

(c) A screenshot of the main view of Magic Tate Ball

Figure 3.1: Site views in stedr and Cooltura and the main view of Magic Tate Ball

The application provides a help site which gives a thorough guide to the user on how to accomplish tasks. This is in accordance with Nielsen's heuristic of help and documentation. The application also helps the user when errors occur, for instance by offering the user a refresh opportunity when the map does not load the sites. The language used in the application is both user-centered and consistent, avoiding misunderstandings and making it easy for the user to understand what different actions entails.

The customer's motivation for creating an entirely new application instead of expanding stedr was mainly because they wished to test personalization systems on users, and thus needed an application focused mainly around the personalization aspect. A more detailed overview of stedr is not included here, as this was done earlier in two student projects related to stedr [**?**].

3.6.2 Cooltura

Cooltura is another application developed under the TAG CLOUD umbrella. The version of Cooltura evaluated here is a demo version, so more developed versions of the application may include more functionality and address some of the issues discussed here.

The content of the stories is somewhat similar to stedr's, with respect to the sites and the stories available. In fact, when clicking to view stories in Cooltura one is directed to the stedr app. However, Cooltura harvests stories from different databases and provides access to much more content about each site than stedr does. When viewing a specific site on Cooltura the user sees a view with a main picture and some text describing the site as seen in **Figure 3.1b**. Like in stedr the user can scroll down to see more content, but unlike stedr the main picture is not static, that is, the user sees progressively less of it as they scroll down. This is more in accordance with Nielsen's point of aesthetic and minimalist design, where less relevant information fills up less space in the user interface. The same is true for the view for a specific tourist attraction. The application does not provide any help site, but the possible user actions are quite few and similar (most of them are about navigating to the desired site), so this might not be a problem.

The application intends to make use of personalization, but the version of the Cooltura app used by the group was not connected to the Cooltura personalization backend [**?**]. The "Anbefalte steder"(Recommended sites)-view in Cooltura now list every site added

in the application (which is three different pilot sites), but the heading suggests that personalization is planned for.

3.6.3 Magic Tate Ball

Magic Tate Ball is an application that presents the user with an artwork based on the input parameters: date, time-of-day, geographical location, live weather data and ambient noise levels [**?**]. The content in the application is the artwork, a description of the artwork and an explanation of why the artwork was chosen. Personalization is the main selling point of Magic Tate Ball. The application uses the input parameters to give the user some control of what content is presented as the user can turn each of these parameters on and off. Even though the application provides an explanation to why an artwork was chosen, how it was chosen remains in the dark (e.g. how to know which input parameters were emphasized in the personalization).

The main task in the application is to present an artwork. This is done by shaking the phone or by clicking a button in the center of the magic ball, see **Figure 3.1c** for a screenshot of this. Another task is to browse all artworks presented to the user, which is done by swiping or clicking on arrows. Providing these two options makes it easy for different types of users, those accustomed to swiping and those who are not. When the application is processing to come up with an artwork to present to the user, it shows the user what the input parameters are. This is in accordance with Nielsen's heuristics on visibility of system status.

CHAPTER 4

TOOLS

This section briefly describes all the tools used for this project, which includes development tools, communication tools and any additional tools.

4.1 Development tools

4.1.1 Front-end

- Ionic [**?**] - Ionic is a front-end UI framework designed to assist the development of hybrid mobile applications. By using this framework it became easy for the team to speed up the design of the interface and test the application both on computers and devices. More details about Ionic can be found in **Section 3.2.2**

- PhoneGap (Apache Cordova) [**?**] - PhoneGap is a framework that enables software developers to automatically wrap HTML5, CSS and JavaScript code into platform-specific code that can run on devices such as iOS and Android phones and tablets. The Ionic framework is based on using PhoneGap for compiling its code. More details about PhoneGap can be found in **Section 3.2.1**

- Android Studio [**?**] - Android Studio is the official IDE for Android application development, it was necessary to have this installed in order to develop our application on android devices. It also provided a way to install various plugins that were useful or needed for the project.

- Node.js [**?**] - Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. This was another tool that was necessary to install, because PhoneGap is built on it. First released in May of 2009, Node.js has been gaining popularity as a server-side platform.

- Gulp.js [**?**] - Gulp is a build tool that is used as a part of PhoneGap in order to automate many common tasks, such as build processes and plugin handling.

- Sass [**?**] - Sass is an extension to CSS which adds functionality such as being able to use variables, nested rules and inline imports. Sass helps keep stylesheets organized, and is fully compatible with regular CSS syntax. Sass is the preferred tool for handling stylesheets in Ionic.

- Proto.io [**?**] - This a prototyping framework aimed for mobile apps, and it allowed the team to make very quick and functional prototypes that were used for user testing, both by the team and by the customer. When discussing design solutions, it was much faster and simpler to make revisions to the prototype than it would be to redesign the application itself.

- Balsamiq [**?**] - Wireframing and mock-up tool which was used to create early mock-ups of the different user interface views. This was used because it allowed the team to quickly make wireframes for the interfaces, and it gives a more professional look than by just making paper prototypes.

- Icomoon [**?**] - This is an application with the purpose of generating an icon font from Scalable Vector Graphics (SVG) files that you upload to it. You can also resize, adjust positions, and set default pixel sizes for the icons. The application turns all the icons into crisp-looking and easy scalable icons. It also automatically generates the HTML and CSS code which you can use to integrate the code into your own application. In this project, Icomoon was used to make the category icons that show the various categories on each story

- FontForge [**?**] - After creating an icon font with Icomoon, FontForge was used to make manual adjustments to the icons themselves. FontForge is an editor with many advanced options for giving icons a smoother look and symmetry.

- Karma [**?**] - Karma is a test runner for AngularJS. With Karma it is possible to write JavaScript tests and run it in different browsers on both desktops and mobile phones. It is easy to run a test for every integration you make. More details about how Karma was used in this project can be found in **Section 8.1**

4.1.2 Back-end

- Docker [**?**] - Docker is an open platform that provides the possibility for system developers to build their application and deploy it to other computers or servers, which can then run the same application, unchanged. The motivation for using

Docker was mainly so the team could run the developed application on SINTEF's own server. More details about Docker is described in **Section 6.10**

- MySQL [**?**] - MySQL is one of the most widely used open source databases. It has many advantages when it comes to scalability and flexibility, and it is well suited for many types of application development. More details about this project's database and its design can be found in **Section 6.9**
- Digitalt musem API [**?**] - The source of the stories in the application was Digitalt Fortalt, and in order to access these it was necessary to use Digitalt museum's API. The API is documented on its website and was incorporated into the project's database. Details about the API and the use of it can be found in **Section 6.7**
- PHPUnit [**?**] - PHPUnit is a framework for writing and performing unit tests on PHP code. More details about how PHPUnit was used in this project can be found in **Section 8.1**
- JUnit [**?**] - JUnit is a framework for doing unit testing in Java.
- DbUnit [**?**] - DbUnit is an extension of JUnit that can be used to test methods accessing a database. Among its advantages is the ability to put the database in a known state before each test, which gives the tester control over what to expect as results of tests that change the contents of the database.
- Mahout [**?**] - Mahout is a project that provides scalable machine learning algorithms. Mahout is primarily focused on algorithms for collaborative filtering, clustering and classification. How Mahout has been used in this project is described in **Section 6.8**.

4.2 Communication tools

- Google Drive [**?**] - Used for creating and sharing documents with the whole group, as well as editing shared documents in real-time. it also made it possible to share all files, whether it was images/diagrams/spreadsheets, or anything else. That was why the team decided to use this as a good solution for sharing all documentation.
- Dropbox [**?**] - Used to share documents between the group and the customer. This tool was used upon request by the customer.

- Facebook [**?**] - Used for discussions and notifications of various things such as meeting times. This was used because Facebook is something everyone was already familiar with, and something that most of the team members use on a regular basis, so messages would be quickly noticed.
- Trello [**?**] - Task management application. This was a handy way to quickly see what needs to be done and what has already been completed, similar to a scrum task board . Because of this the team found it to be a good tool to use, as it speeds up the process of managing work tasks and gave a better overview of the progress.

4.3 Additional tools

- GitHub [**?**] - Used for making a code repository to be shared by the group while developing the system. The customer requested the use of GitHub, and it was also used because it seemed like the easiest way to share and implement code, as well as sharing the code with the customer.
- Draw.io [**?**] - This tool was the primary way of making diagrams and models, such as use-cases and WBS chart. This tool was chosen for this because it provides a lot of templates for different types of graphs, and as everyone uses the same tool for all diagrams the report achieves a consistent style for every diagram.
- Ganttify [**?**] - Converting Trello boards into Gantt Charts, makes the process of creating a gantt chart and milstone plan easier and faster.
- LaTeX [**?**] - LaTeX is a document preparation system. Used to produce the final version of the report. Chosen because it provides control over sectioning, cross-referencing, tables and figures. The code for the report was hosted by GitHub to ease cooperation.
- TeXstudio [**?**] - TeXstudio is an integrated writing environment used to create LaTeX documents. This tool was chosen because it has a user interface which simplifies the work.

CHAPTER 5

PROJECT MANAGEMENT

The following chapter describes how the project was planned and includes risk management, a description of meetings with involved parties, delegation of work tasks and responsibilities, a work breakdown, time management and quality assurance.

5.1 Risk management

Risk management includes planning and handling all the various potential risks to the project.

The risk analysis contains a list of possible occurrences that could be harmful to the project. Provided for each risk is a short description, an estimated likelihood that the risk will happen, an estimated impact to the project if it happens, the importance of the risk, a preventive action to try to avoid the problem and a remedial action if the problem were to occur. Likelihood and impact estimates were rated on a scale from 1 to 9, with 9 being the highest, and the importance was calculated by multiplying likelihood with impact. The risk list is sorted by the importance value, thus the risk to be most aware of at each stage of the development process was at the top of the list. Risks were updated regularly and changes were made when new issues were discovered.

Table 5.1 describes three of the most important risks in the risk analysis. The complete risk analysis is located in **Appendix B**.

L: Likelihood (1-9)
I1: Impact (1-9)
I2: Importance (Likelihood * Impact)

Table 5.1: Risk list excerpt

Description	L	I1	I2	Preventive action	Remedial action
Underestimate the time planned to use for assignments	8	8	64	Estimate a little higher. Continuous meetings. Continuous status update on tasks	Extra work hours and help each other. Have a clear prioritization of tasks so that some less important tasks can be delayed if needed.
The group does not deliver updated information on the report and cannot maximize the quality of the feedback obtained from the supervisor	6	7	42	Always make sure the report meet the demands upon delivery	Ask concrete questions to the supervisor or other competent acquaintances of the group members
An issue in the code that is not understood or can not be fixed.	5	8	40	Comment on the code, and talk with each other about the work done on the code	Get help from the supervisor or other people involved.

5.2 Meetings

The group had many meetings with parties that were involved with the project and/or the application. The three main types of meetings documented were group meetings, customer meetings, and supervisor meetings.

It was logical to have one-week sprints since the customer meetings were on a weekly basis. Since the team was a group of seven it was impossible to always work together. Because of this, the regular scrum meetings were beneficial to share and discuss progress. During each meeting, a summary was written to document what the group members had done since last, what issues had occurred, and any other important notes. An example of a group meeting report can be found in **Appendix F**.

The customer requested weekly meetings, and to make the most out of these meetings,

the team planned an agenda ahead of each meeting. This was done to improve the structure of the meeting and to give the participants time to consider the issues on the agenda, which in turn was meant to increase the benefits of the meeting and increase the likelihood of making decisions. During each meeting, a summary was written as well. An example of a customer meeting report can be found in **Appendix F**.

The supervisor requested bi-weekly meetings which were used to discuss the report. The supervisor read the most recent version of the report in advance of each meeting and gave feedback on what needed to be changed. Because of this, after each meeting the team kept a list of feedback from the supervisor of issues that should be handled before the next supervisor meeting. Status reports were also made and sent to the supervisor before each meeting, and an example of a status report can be found in **Appendix D.1**

5.3 Scrum team and roles

The role delegation in the team is detailed in **Table 5.2**. The delegation of roles was primarily based on personal interest and motivation. The tasks were divided up into main responsibility areas for back-end and front-end before assigning people to each one. However, this was only a guideline for main responsibilities, and the group members had to be flexible and work with tasks outside of their main areas.

5.4 Work breakdown structure

A work breakdown structure (WBS) is a decomposition of the project, and its goal is to break down each part of the development process into manageable parts to ease the planning and execution of the development. Each element in the diagram can be a product, data, service or a combination of the three. One of the benefits of detailing a project this way appears when doing cost estimation and scheduling the team around the project (i.e. should ease the project planning and help allocate the team's resources).

The WBS shows a hierarchical decomposition of the project phases and its components. Each main phase is at a top-level and will outline the generic parts of the software development processes. The way the WBS is developed is by starting with the end objective and subdividing each main part into manageable components in terms of size, complexity and duration. Each sub objective is to follow the 80 hour rule. This means that a subtask is

not to exceed 80 hours in magnitude. The WBS for this project is shown below in **Figure 5.1**.

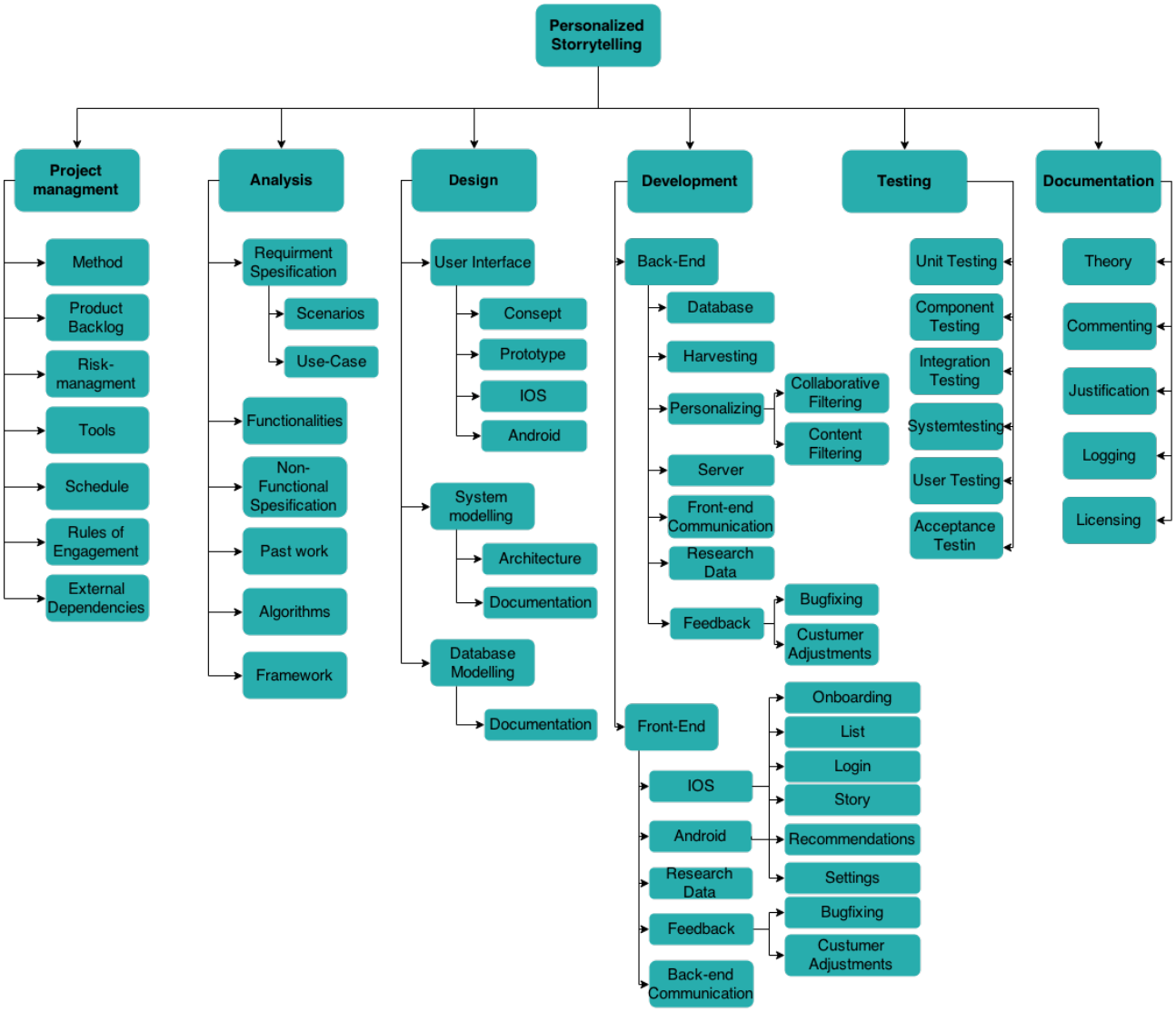


Figure 5.1: Work breakdown structure

5.5 Project milestone plan

Milestones are used as tools in project management to give the team some clear and specific goals to work towards as the project timeline moves ahead. There are several milestones throughout the project, some are large milestones, like the alpha and beta versions

of the software. There are also milestones related to the project report, like the midterm submitting and final delivery. Milestones can add value to the project scheduling when used in the right manner and when setting realistic goals. Components that are important for the milestone plan are; key dates, key deadlines and external deliveries. The team used a combined Gantt chart with milestones noted for better visualization, and to better allocate resources for meeting the milestone goals. The Gantt chart is presented in **Figure 5.2**.

The planned deliveries to the customer are the following dates.

- **20.02.15 - First prototype**
  A prototype should be finished by this date. Paper or mockup would suffice
- **27.02.15 - Second prototype**
  The next iteration of the prototype should be presented in Proto.io [**?**]
- **17.03.15 - Alpha**
  First working software should be ready for a customer test
- **10.04.15 - Beta**
  A feature ready version of the application should be ready for the customer
- **01.05.15 - Final product**
  A polished and well tested application and ready to deploy back-end should be ready for delivery

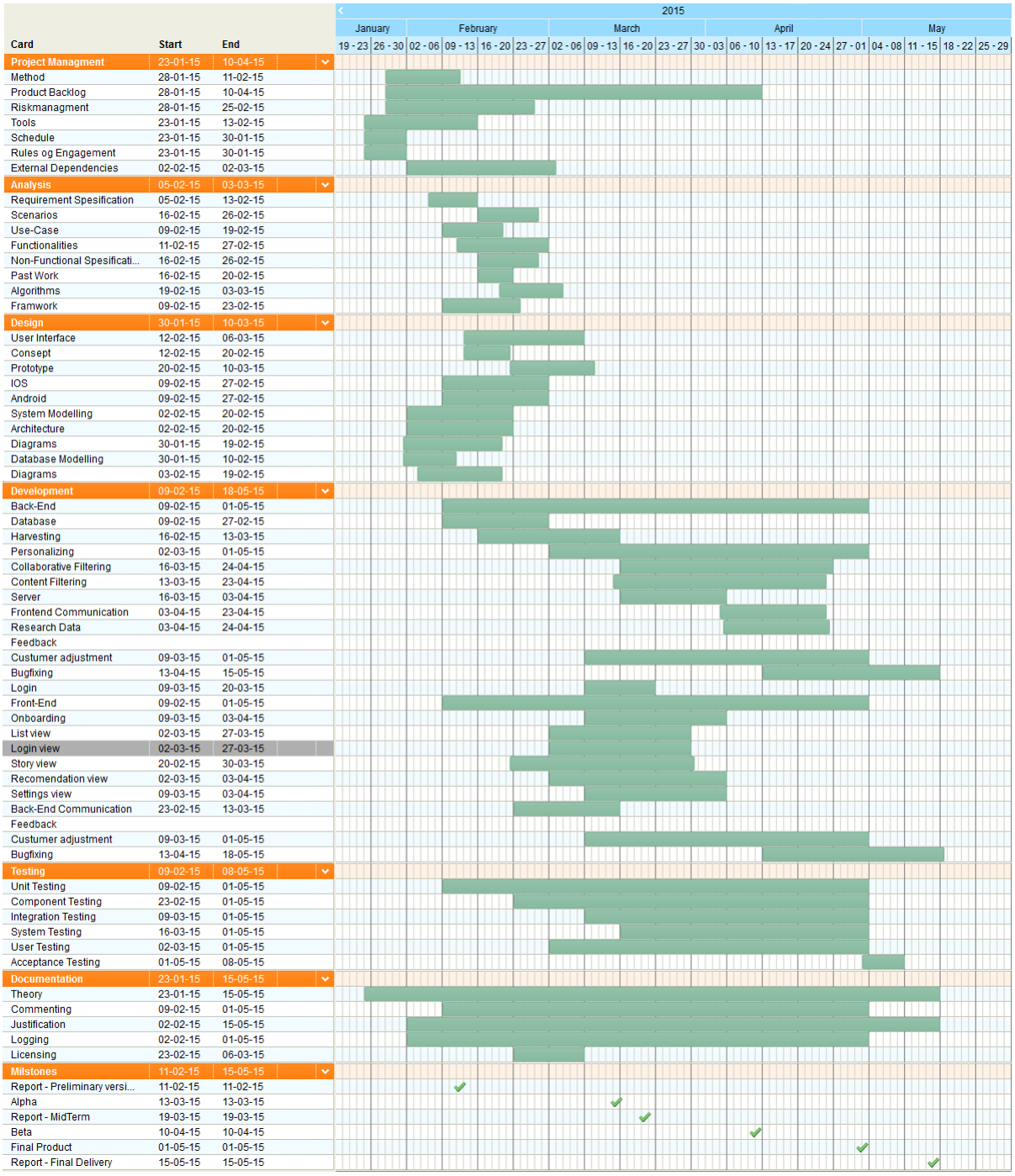
5.6 Burn down

A burn down chart is a visual representation of the outstanding work(Appendix D.2) relative to the projects remaining time. Its visual representation is outlined in a graph where the x-axis represents the timeline where each point on the graph is one sprint, and the y-axis represents the estimated hours needed to finish the project. This is useful to have a better understanding of the rate that the project is moving and it indicates whether or not one will meet the project's estimates.

forklar hvordan man kom fram til dette?, legg hele gantt til i appendix?

Skrive om activity plan

Skal vi sitere her? http://idi



Card	Start	End
Project Managment	23-01-15	10-04-15
Method	28-01-15	11-02-15
Product Backlog	28-01-15	10-04-15
Riskmanagment	28-01-15	25-02-15
Tools	23-01-15	13-02-15
Schedule	23-01-15	30-01-15
Rules og Engagement	23-01-15	30-01-15
External Dependencies	02-02-15	02-03-15
Analysis	05-02-15	03-03-15
Requirement Spesification	05-02-15	13-02-15
Scenarios	16-02-15	26-02-15
Use-Case	09-02-15	19-02-15
Functionalities	11-02-15	27-02-15
Non-Functional Spesificati...	16-02-15	26-02-15
Past Work	16-02-15	20-02-15
Algorithms	19-02-15	03-03-15
Framwork	09-02-15	23-02-15
Design	30-01-15	10-03-15
User Interface	12-02-15	06-03-15
Consept	12-02-15	20-02-15
Prototype	20-02-15	10-03-15
IOS	09-02-15	27-02-15
Android	09-02-15	27-02-15
System Modelling	02-02-15	20-02-15
Architecture	02-02-15	20-02-15
Diagrams	30-01-15	19-02-15
Database Modelling	30-01-15	10-02-15
Diagrams	03-02-15	19-02-15
Development	09-02-15	18-05-15
Back-End	09-02-15	01-05-15
Database	09-02-15	27-02-15
Harvesting	16-02-15	13-03-15
Personalizing	02-03-15	01-05-15
Collaborative Filtering	16-03-15	24-04-15
Content Filtering	13-03-15	23-04-15
Server	16-03-15	03-04-15
Frontend Communication	03-04-15	23-04-15
Research Data	03-04-15	24-04-15
Feedback		
Custumer adjustment	09-03-15	01-05-15
Bugfixing	13-04-15	15-05-15
Login	09-03-15	20-03-15
Front-End	09-02-15	01-05-15
Onboarding	09-03-15	03-04-15
List view	02-03-15	27-03-15
Login view	02-03-15	27-03-15
Story view	20-02-15	30-03-15
Recomendation view	02-03-15	03-04-15
Settings view	09-03-15	03-04-15
Back-End Communication	23-02-15	13-03-15
Feedback		
Custumer adjustment	09-03-15	01-05-15
Bugfixing	13-04-15	18-05-15
Testing	09-02-15	08-05-15
Unit Testing	09-02-15	01-05-15
Component Testing	23-02-15	01-05-15
Integration Testing	09-03-15	01-05-15
System Testing	16-03-15	01-05-15
User Testing	02-03-15	01-05-15
Acceptance Testing	01-05-15	08-05-15
Documentation	23-01-15	15-05-15
Theory	23-01-15	15-05-15
Commenting	09-02-15	01-05-15
Justification	02-02-15	15-05-15
Logging	02-02-15	01-05-15
Licensing	23-02-15	06-03-15
Milstones	11-02-15	15-05-15
Report - Preliminary versi...	11-02-15	11-02-15
Alpha	13-03-15	13-03-15
Report - MidTerm	19-03-15	19-03-15
Beta	10-04-15	10-04-15
Final Product	01-05-15	01-05-15
Report - Final Delivery	15-05-15	15-05-15

Figure 5.2: Gantt chart

Our estimates were done on the basis that each team member would have a workload of about 20 hours each week.

The burn down estimate done at the start of the project can be found in **Figure 5.3** below.

Dette ble sagt i forelesning. Men på nettet står

The final burn down of the project can be seen in **Figure 7.1**.

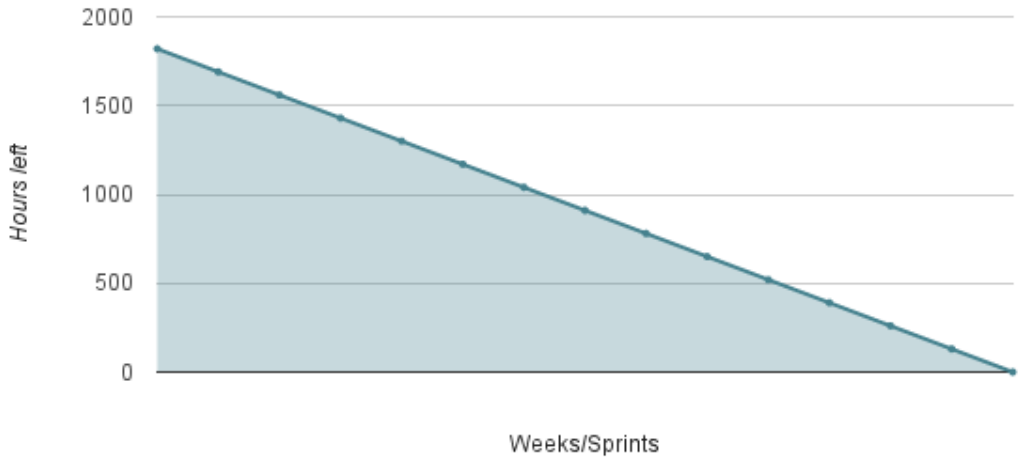


Figure 5.3: Estimated burn down chart

5.7 Quality assurance

According to Sommerville, quality assurance is "the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process" [**?** , p.652]. In large systems designed to be used in a long term perspective, quality documentation is important. However, in this project a smaller system was developed and Sommerville notes that a more informal approach can then be applied, focusing on "establishing a quality culture" [**?** , p.654] within the development team.

Therefore, this section will describe four features considered important by the group to establish such a quality culture and thus improving the quality of the product, namely group interaction, version controlling, code quality and interaction with the customer. Risk management and testing are also considered important aspects of quality assurance, and these are discussed in separate parts (see **Section 5.1** for risk management and **Chapter 8** for testing).

5.7.1 Group interaction

As was noted in **Section 3.3.3**, an important part of the scrum methodology is the close collaboration between the team members. Scrum provides some events to enhance this collaboration, for instance the daily scrum meetings, and some artifacts, such as the sprint backlog. These features of scrum were used by the group and helped create a framework for the process development. However, scrum does not define how the group should interact, and the group interaction consisted of more than the methods provided by scrum, for instance when doing sessions of collaborative work.

In order to ensure that the scrum events and the interactions external to these events would create the desired quality culture, the team discussed and agreed upon some basic rules of engagement for the project (see **Appendix A**). These rules specified how the team should create a quality process in order to create a quality product, for instance by setting ground rules for communication between the group members. The tools described in **Section 4.2** were used to facilitate the implementation of the rules. In addition, meeting minutes from every meeting were made so that every group member would be aware of the status of the project even if they were not present at the meeting.

The division of the group described in **Section 5.3** meant that a member of the front-end part would have more detailed information about what other front-end developers were doing than what individual members of the back-end part were doing, and vice versa. However, when important decisions were to be made in one part of the project or important problems had to be solved, both parts would be involved in the discussion, even if the decision did not affect their part of the project directly. An example of such a decision was the choice of colors to use in the user interface.

5.7.2 Git and version controlling

The code for this project is hosted by GitHub, a tool described in **Section 4.3**. GitHub uses the version control system Git, which makes development easier by allowing multiple local branches and thereby giving users the opportunity to try out code before committing it to the master branch. The master branch is the main branch and should only include stable code. The group chose to create two repositories, one for the back-end part of the project and one for the front-end part, since the group was divided in this way as well. Doing it this way made keeping track of branches and issues on GitHub clearer since

these often would be at a level of detail only relevant to the developers in that part of the project.

5.7.3 Code quality

In order to ease the cooperation on the code for the project, the group followed some general guidelines. These guidelines were:

- Use descriptive names on variables, classes, folders and other elements
- Comment code
- Divide the code into logical units (See **Section 6.2** for an example of what this entails.)
- Make sure the code units are not too large

5.7.4 Customer interaction

In **Section 3.3.3** it was noted that the project was flexible and likely to have changes in the requirements. This meant that good customer interaction was critical, so that the group and the customer would be in agreement of what was expected of the end product. The customer meetings are described in **Section 5.2** and apart from these meetings, communication with the customer was also done by mail and by a shared Dropbox folder. In addition, the customer were by their request watchers of the code on GitHub.

Making decisions and coming to an agreement with the customer on key issues was considered important to push the project forward. The meeting minutes written after each meeting allowed the customer to know if all the participants had a similar understanding of the issues discussed and the decisions made. Communication by mail was mainly used for rescheduling of meetings and by the customer to give additional information to the group.

Table 5.2: Role delegation

Role	Responsible	Details
Product Owner	Jacqueline Floch (SINTEF) Shanshan Jiang (SINTEF)	Responisible for convaying the vision of the product to the scrum team and prioritize the featsure list for the product. [**?**]
Group supervisor	Soudabeh Khodambashi	When Soudabeh was on sick leave in March, Nina M. Smørsgård filled in for her as supervisor for the group
Main back-end staff	Hanne Marie, Eivind, Kjersti, Audun	
Interaction between server modules	Hanne Marie	This involves making the different server components communicate and work together, such as the database, personalization module, Docker file, etc.
Personalization	Eivind	Involves the procedure for taking a user's context information and preferences, and based on this choose which stories to present to the user. Also involves collaborative filtering to select stories for a user based on the preferences of other users.
Test responsible	Kjersti	Involves creating a test plan for each kind of test (unit, usability, integration, etc.) and documenting the results, as well as assuring test quality and that the tests are actually conducted properly.
Database Manager	Audun	Involves setting up and managing the database. Manage which elements to save in the database and how to present the data upon request.
Main front-end staff	Ragnhild, Roar, Espen	
iOS responsible	Ragnhild	Involves developing and giving the app a look and design that feels intuitive and follows design conventions according to iOS systems. This is both on a technical and design level.
Android responsible	Roar	Involves developing and giving the app a look and design that feels intuitive and follows design conventions according to Android systems. This is both on a technical and design level.
Framework responsible	Espen	Involves developing various parts of the UI, and researching the framework to ensure that the front-end design can be made and is made according to the team's and customer's expectations.
Additional roles	Roar, Kjersti, Espen	
Scrum Master	Roar	Project leader, responsible for arranging meetings, delegating work tasks, and overseeing the general progress of the project. Facilitate the communication and cooperation between the group members.
Customer relations	Kjersti	This includes all communication with the customer, as well as other authorities that are a part of the project.
Report management	Espen	This involves overseeing the report work and ensuring that all components of the report are in place. Deliver the report within deadlines and make changes as needed and in accordance with advice from supervisor.

CHAPTER 6

DESIGN AND ARCHITECTURE

To get a brief overview of the complete product and its required parts, the design and architecture are presented in this chapter. This is not meant to give a complete understanding of the system, but rather an overview on how the different parts of the product work together to give a pleasant user experience.

6.1 Architecture

The overall architectural design of the system was made to achieve a rough mapping of what needed to be done in terms of actual programming. The architecture focuses heavily on interactions between the different instances of the systems without going into the specific details on how this is done. To illustrate the architecture, two different views were created. One showing only the components, and the other showing the processes in the different components. The overall system structure can be seen in **Figure 6.1**, this shows the four different components and how they interact. As shown in this diagram, the system is created as a client-server model, where the mobile application constitutes the client, which communicates with the PHP back-end server. The server is connected to a database, and Digitalt fortalt provides an API to retrieve stories from.

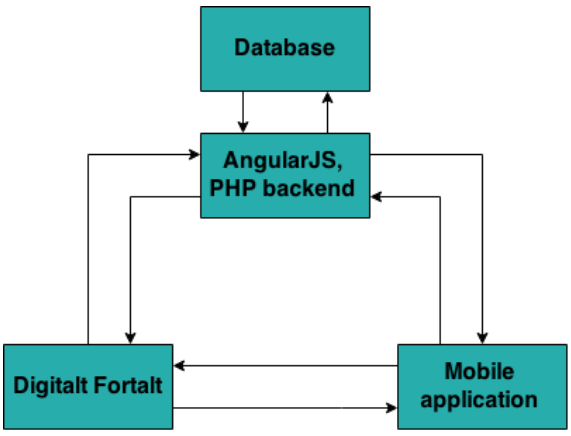


Figure 6.1: Diagram of the overall system structure for this project.

Figure 6.2 shows a diagram presenting the architecture for the application. As seen in the legend, the square boxes represent individual components or processes. The boxes with oval corners represent compound processes or larger parts. These are mainly shown because they give an overview of which processes belong to and will be performed by which part of the system. Double lined boxes are external sources which provide an API. Lastly, arrows indicate information flow.

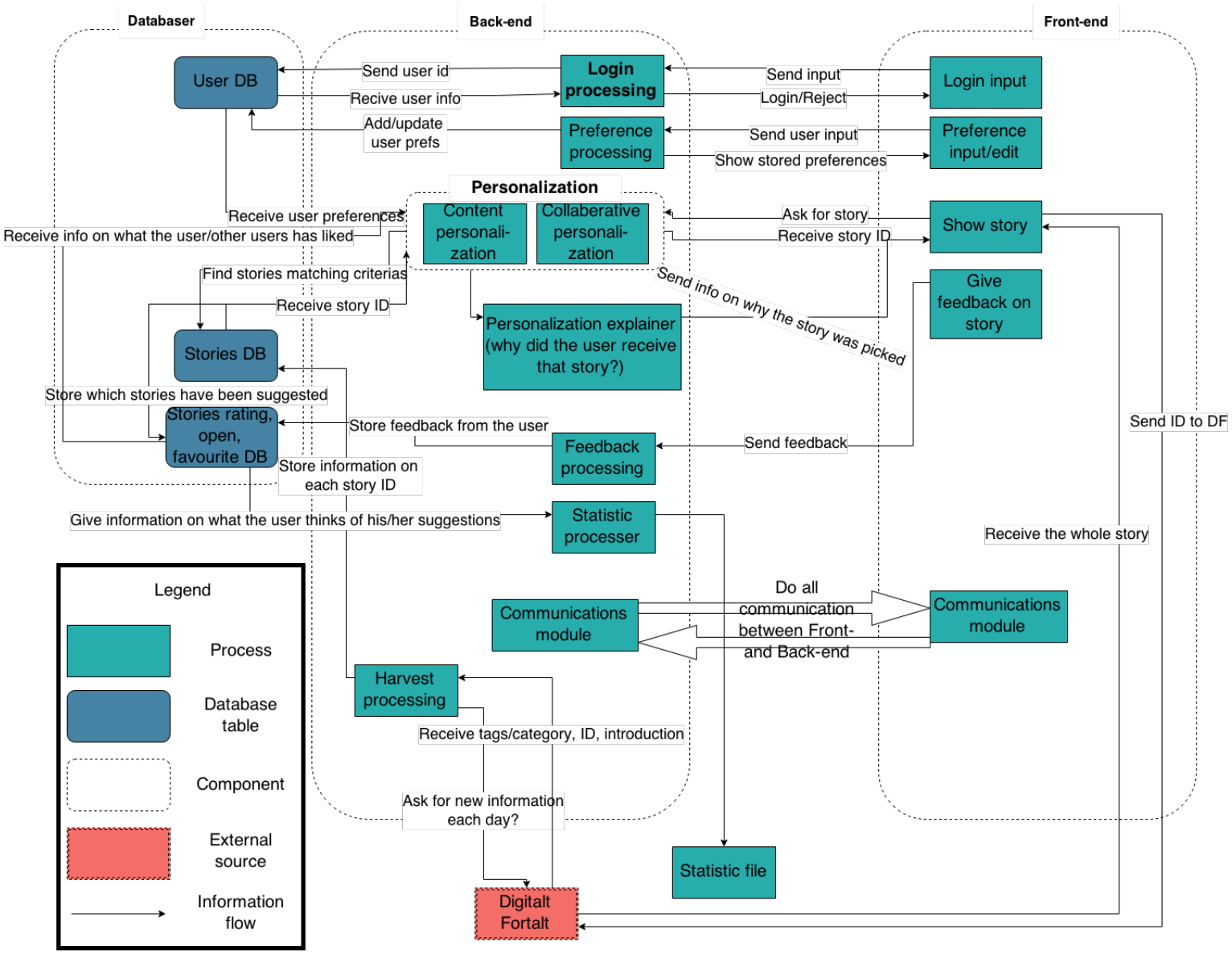


Figure 6.2: Diagram of the architecture for this project.

As seen in **Figure 6.2** the application structure is divided into three components; database, back-end and front-end. The front-end component is responsible for displaying information to the user. It also handles user input, which it conveys to back-end for processing (see **Section 6.4**). How front-end is built up and what is displayed to user can be found in **Sections 6.2** and **Sections 6.3**, respectively. The back-end section is responsible for handling requests from front-end, returning information to front-end and for harvesting the stories stored in the database from Digitalt fortalt. **Section 6.5** describes the back-end structure. The database component stores all the persistent information related to users and stories. The database is described in **Section 6.9**.

It is a difficult task to model a whole system in an accurate way, and while the architecture diagram gives an overview, it can not give much insight into the complexity of each process. This is further complicated by the fact that in the startup phase of the project there are a great number of unknowns. Both complexity and requirements are subject to change. As such, the diagram should only be used as a guide for understanding the composition of the complete system.

6.2 Front-end structure

The front-end of the application was designed by using Ionic and AngularJS and this section will describe how a system made in AngularJS is structured.

AngularJS provides templates to create systems based on a Model-View-Controller (MVC) architecture, and also provides a variety of components to assist in making the programming more effective and speeding up the application. It is essentially another layer of abstraction above writing regular JavaScript. These are some of the main concepts that are used to create an Angular application:

Model

Manages all the data that can be interacted with by the user. The model also keeps the views up to date and can be manipulated by controllers.

View

The interfaces that the user can see. This means some form of visual representation of the model.

Controller

The logic and behavior of the views. The controllers also make changes to the model.

Directive

Special functionality applied to the DOM elements, you can create your own directives as well as use the numerous ones provided by AngularJS itself. In ”Vettu hva?”, several of these were used, such as directives that call on some function when DOM elements are clicked on, or directives that show or hide parts of the DOM based on some condition.

Module

A module encapsulates a part of the application. Instead of having a single ”main” function that holds the application together, Angular applications normally have multiple modules that work together. The benefit of this is that the application is decomposed into logical parts that can be reused, loaded, and tested in any order. In ”Vettu hva?”, each controller is its own module. The part of the program that communicates with the back-end is also encapsulated in a module. Furthermore, there is a module that starts up the application and binds together views and controller into different states.

Scope

The scope contains all the elements that the application currently has access to. It can be viewed as a container that stores the current model, and so if a controller or directive is going to modify or access the model, this has to be done by using the scope.

Service

A service contains ”global” logic that is accessible to the entire application. In ”Vettu hva?”, this is used in the module that contains the communication with the back-end. This module has three different services, which respectively gives the application access to user data, story data, and requests from front-end to back-end.

6.3 User interface

It was very important to the customer that the application should be easy to use (**Section 2.2**). The elements added to the user interface were thus only things that were absolutely needed in order to satisfy the customer’s requirements. The customer also wanted it to

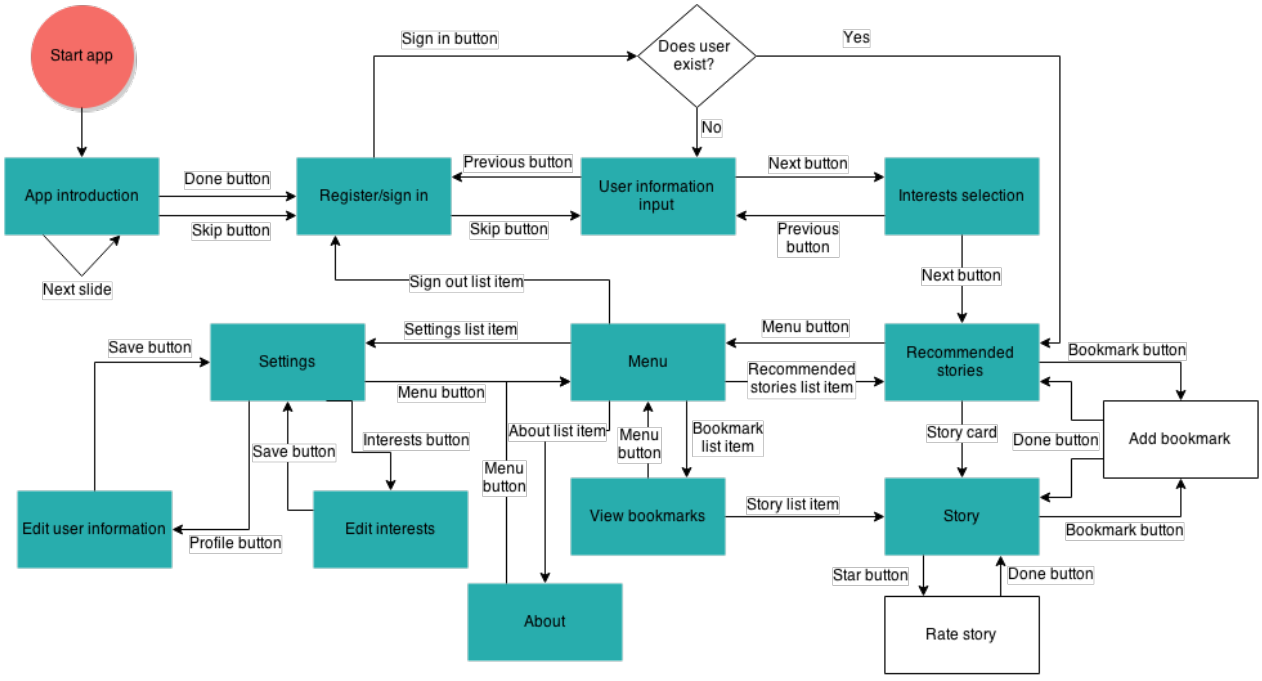


Figure 6.3: Diagram of the flow between views in the user interface.

”feel native”. As it would be very time consuming to make two very different designs for Android and iOS, the design of the app was made quite different typical Android or iOS. This way it would not look out of place on either platform, and it would not be dated as easily when new operative system versions come out. For example, the hamburger menu pattern can be found on both platforms. This also made the interface simpler as the functions that are not used as often can be out of sight. It helped the focus on the main function, exploring stories.

The usability of an application is not just about the individual views, it is also about the structure and flow of the app. **Figure 6.3** explains the overall flow between all the different views in the application. The green boxes represent views, while the white ones represent modals placed on top of the view the user came from. The text of the arrows explain what the user clicked in the view the arrow comes from, and the arrow points to which view this action leads to. The functionality of the more complex views will then be explained in further detail. The rest of the views can be found in the user manual (**Appendix H**).

Recommendation view

This view displays the stories that should be the most relevant to the user. The user can browse them by swiping through them or by clicking the left and right

arrows. When tapping on a card the detailed view of the story will be displayed. The bookmark icon in the top bar opens a modal for adding a bookmark.

Figure 6.4: Recommendation view

Story view

This view displays the chosen story in detail. There is a box which displays the media files associated with the story. The tabs above it will depend on which media types the story contains. Videos will be displayed by default if there are any, as they can be a major component of the story which should not be hidden in another tab. When tapping on a video or image it will be displayed in fullscreen mode. The user can give feedback on the story either by tapping the stars on the bottom part of the view, or by tapping the star icon in the top bar which will open up a modal. The modal will ask the user to rate the story, and the user can exit it by tapping "Ferdig" or by tapping outside of the modal. Tapping the bookmark icon will make the same modal as in the recommendation view appear.

Figure 6.4: Story view

6.4 Front-end - back-end communication

Communication between front-end and back-end was handled using HTTP post requests. AngularJS $http is a core service for reading data from remote servers, which is called every time the application needs to add, retrieve, update or delete data. When an HTTP request is made, four fields are set: method, headers, URL and data. The method field determines the HTTP request method, which in this application is set to post, and the headers field sets the content type to JSON. The URL is the location of the remote server script that handles HTTP requests. In the data field the action to be executed is specified, in addition to any data needed to perform the desired action.

Each HTTP request is managed by the same back-end PHP script. This script decodes the HTTP request, determines which action to perform and executes it. When the script has finished executing, a JSON response is returned to front-end with the desired data.

”Get story” is an example of a request. When a user wishes to read a story front-end sends the JSON formatted paramaters userId, storyId and request type, in addition to the previously mentioned fields, as an HTTP request to back-end. Back-end retrieves story information from the database and Digitalt fortalt, which is then returned as a JSON formatted response to front-end. Other request examples are ”Add user”, ”Edit user” and ”Get recommended stories”.

6.5 Back-end overview

The back-end design is divided into five sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of the application. The value of such a separation is that it simplifies the development and maintenance of the application code. By splitting the code into separate parts the system is more logically structured, and cleaner code is achieved. The result is as an example that SQL related code is maintained by one part of the system, while HTTP processing is executed by another. Below is a description of the different back-end sections.

Models

Consisting of the classes storyModel, userModel and preferenceValue. The models are used to temporarily hold information about a story, a user and a user's preference value for a story to be utilized by other files. Information is either retrieved from the database, sent from front-end, harvested from Digitalt museum's API or a combination of these. These models also contain formatting methods, which makes it possible to return story or user information to front-end.

Database

This section contains the classes dbStory, dbUser, dbHelper and harvesting. The db classes are responsible for accessing the database. dbStory contains methods for adding or retrieving story related information from the database and dbUser is responsible for user related information. dbHelper consists of more general methods and is the class which establishes a connection with the database. The harvesting script is responsible for collecting all database stories from Digitalt museums API and adding stories to or updating stories in the database (see **Section 6.7**).

Personalization

Consisting of the classes computePreferences and runRecommender. This section

computes preference values for each Digitalt fortalt story in the database for each user. runRecommender is also responsible for initializing the Mahout recommendation engine when a user's preference values have been calculated.

Recommender

This section consists of the Java code and recommender.jar file which make up the Mahout recommender (see **Section 6.8**).

Requests

Contains the controller script, which receives and handles front-end HTTP requests and returns JSON responses (see **Section 6.4**).

Figure 6.5 depicts the different back-end classes introduced in the previous paragraphs and give an overview of their dependencies.

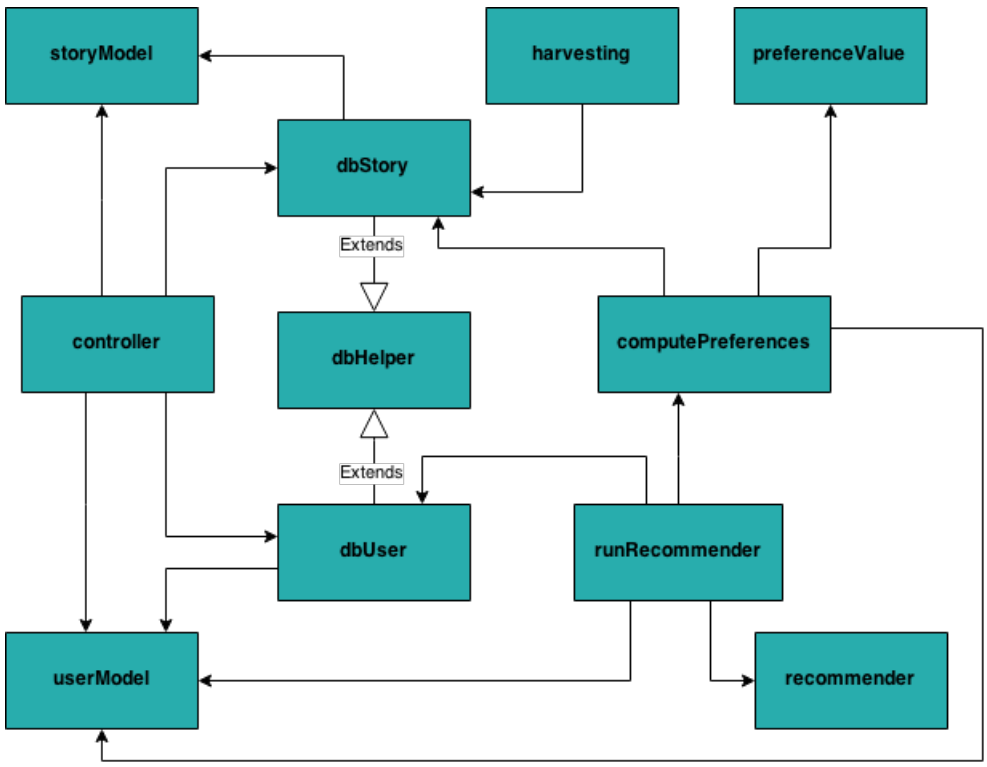


Figure 6.5: Class diagram of the overall back-end structure

6.6 Category mapping

There are 31 subcategories present at Digitalt fortalt. Each story can have 0 to 31 subcategories attached to it. To achieve content-based filtering the user has to select some categories of interest. To make it easier for the user, these subcategories are divided into nine main interest categories. Some of these categories were already predefined in a TAG CLOUD document provided by the customer, while the rest was decided in discussions with the customer. The subcategories were put into the category interest field that fit them best, with some subcategories being in several category interests. In **Table 6.1** this mapping is illustrated. It is obvious that some category interests will have more stories, and some subcategories such as history contain more stories than most others. It was intended to create nine category interests that contain roughly the same amount of stories. However, when a subset of stories was chosen this intention did not hold true as can be seen in chapter **Section 6.7**. Furthermore, the category interests needed to be distinct while still encompassing all the subcategories.

Table 6.1: Category mapping performed to facilitate, and simplify content-based filtering. Each subcategory is assigned to one or more category interests.

Archeology	Arkeologi og forminne
Architecture	Arkitektur
Art and design	Bildekunst, dans, design og formgjeving, film, fotografi, media, teater
History	Historie, historie og geografi, kulturminne, sjøfart og kystkultur, språkhistorie
Local traditions and food	Bunader og folkedrakter, dans, fiske og fiskeindustri, fleirkultur og minoritetar, Hordaland, kultur og samfunn, kulturminne, musikk, Rallarvegen, samer, sjøfart og kystkultur, språk, tradisjons- mat og drikke
Literature	Litteratur, teikneseriar
Music	Musikk
Nature and adventure	Fiske og fiskeindustri, naturhistorie, sport og friluftsliv
Science and technology	Fiske og fiskeindustri, fotografi, ”kjøretøy, bil og motor, veitransport”, media, ”natur, teknikk og næring”, ”teknikk, industri og bergverk”, skip- og båtbygging

6.7 Digitalt museum's API and harvesting

The content of the stories displayed in the application was collected from Digitalt fortalt. At the request of the customer the stories collected are limited to the areas Nord-Trøndelag and Sør-Trøndelag. The reason for this was that it would increase the likelihood of users reading similar stories, and thus make it easier to evaluate the personalization. The API [**?**] used to retrieve stories belongs to Digitalt museum. This API enables search through data from Digitalt museum, displaying pictures, and provides access to an XML representation of available objects. Digitalt fortalt is established on the same technical platform as Digitalt museum [**?**]. This makes the integration better between the two and the remaining services in Norvegiana. Norvegiana is a data model, database and a web service with the purpose of making cultural heritage information more accessible [**?**]. Example services available in Norvegiana are Digitalt museum, Digitalt fortalt, Arkivportalen and Musikkarkiv.

Some info, such as the categories for each story, are harvested and stored in the database. This is mainly to facilitate personalization. The actual story and the related media are fetched from Digitalt fortalt at the request of a specific story from front-end. At the time of writing the number of harvested stories is 169. This may vary as the harvesting is done each day. The stories are distributed unevenly over the nine categories; both the history and local traditions categories have over one hundred stories connected to them, while the literature category only have two stories connected to it. Media formats distribution also varies, with only fifteen stories having sound and both picture and video included in over one hundred stories.

6.8 Personalization

Users receive recommended stories by means of content-based and collaborative filtering, as described in **Section 3.4**. The system gathers information about the stories and the users, and feeds this information into the recommender engine Mahout which produces a list of recommendations for the specified user. The details of Mahouts inner workings will not be presented here. What will be described is how the input data delivered to Mahout is gathered and produced by our application, what methods of Mahout is used and when they are used, and how the resulting list of recommendations is dealt with.

Both content-based and collaborative filtering rely on giving a numerical value that express how much a user likes a story. We call this a preference value. In our application this value is computed by combining several measures representing different types of user feedback. These are: the rating a user have given to a story, the categories the user has expressed an interest in, the number of times a story has been recommended to the user, the number of times the story has been opened and the number of times a story has been put in the to-be-read list. Weights are assigned to the measures to differentiate between what impact they should have on the preference value. Rating is considered the most important user feedback, since this value is given to a story by direct user action. The other measures are either less connected to a specific story or more intangible. When computing recommendations, preference values for all stories for the user in question are calculated first.

Each time Mahout is run, up to ten recommendations are inserted in the database. How these are chosen depends on a number of factors. Since a requirement from the customer was to include false recommendations, one of the recommendations is always picked from the lower parts of the recommendation rankings. The other stories are picked from the top of the rankings. Depending on two factors some top recommendations may be skipped for a given run. The first of these factors is that rated stories are not recommended again to the user (only an issue for content-based filtering). The other one is whether the list of recommendations should be added to the existing list of recommendations browsed by the user in the recommendation view or not. When adding to the existing list, recommendations should not be repeated. Adding to list happens when the user draw near the end of the current list of recommendations in the recommendation view, while new recommendations are computed when the user rates a story, changes its category interests or leaves the application (to make sure the recommendations are fresh next time the user enters the application).

Mahout uses a data model and a similarity model to compute recommendations. The data model is a collection of triplets, where each triplet consists of a user ID, a story ID and a preference value. The selection of which preference values to include is different in content-based and collaborative filtering. The similarity model is also different for content-based and collaborative filtering. A description of these differences follows in the subsequent sections.

6.8.1 Content-based filtering

When doing content-based filtering, the input data model consists of a collection of all the preference values for the user we are making recommendations for. This means that the size of the model will be equal to the number of stories harvested from Digitalt fortalt. To make recommendations Mahout also need measures describing how similar stories are. Similarities between stories are computed based on the subcategories found in the second column in **Table 6.1** using the cosine similarity formula:

$$similarity = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| \cdot \|\vec{y}\|} \tag{6.1}$$

The dot product in the numerator is in our system equivalent to the number of common subcategories for two stories, while the denominator is the product of the square roots of the number of subcategories for each of the two stories. If two stories have exactly the same subcategories the similarity value will be 1, and if they do not share any subcategory the value will be 0.

These similarities are computed and put in a comma-separated values (CSV) file after the harvesting is done. This file is read before the recommendations are computed and the similarity values are put in the appropriate Mahout objects. Mahout then uses the data model and the similarity model with similarity values for all possible pairs of stories to produce a ranked list of recommendations. This list is enhanced with an explanation of why a story was recommended, using a Mahout method that returns the stories most influential for a given recommendation.

6.8.2 Collaborative filtering

Collaborative filtering uses a data model with multiple users. Only the stories a user has rated will be included in the data model since this is the most precise user feedback. In our application collaborative filtering is done by combining two different methods provided by Mahout, namely item-based recommendation and user-based recommendation. The item-based approach creates a similarity model based on similarity between items in the data model using a logarithmic Likelihood function, while the user-based approach creates a similarity model based on similarity between users in the data model using Pearson correlation. To prevent dissimilar users from affecting the recommendations in the user-based approach, a threshold value for the similarity is set. The two lists of recom-

mendations are produced and then merged, assuming both have content to get one list of ranked recommendations. If only one of the lists has content then the predicted rating for the story is used to decide what story to serve next.

As was noted in **Section 3.4** collaborative filtering requires a large amount of data to make recommendations. Since only rated stories are included, collaborative filtering cannot be run when a new user starts using the application as the user will not be part of the data model. The criteria for using collaborative filtering needed to be met is that the user has rated at least ten stories which at least ten other users also have rated. In the event of too few collaborative story recommendations content-based filtering is used.

6.9 Database design

The database was designed with the goal of facilitating the recommendation of stories to users. The data model underpinning the database is visualized in the ER-diagram in **Figure 6.6**. User and story are the central entities in this diagram, since the goal of the application is to connect users to stories. Both of these entities has a number of attributes describing the entity. The central relationship in the diagram is the recommendation-relationship, where a user and a story is connected. This connection is described by some additional attributes, such as rating, tag (which is the same as a bookmark) and state.

To make the right connections between user and story, some attributes describing the user and the stories are necessary to store in the database. For instance, in order to make recommendations based on nine predefined categories, every story is mapped from sub-categories gathered from Digitalt fortalt to one or more of the nine categories (see **Section 6.6**). A user is connected to one or more of these categories by the setting of personal preferences. The changing state of recommended stories is also recorded, partially to make better recommendations and partially for research purposes.

As described in **Section 2.1.1** a requirement for the project was to provide some research data to the customer. To do this some data about the use of the application is stored, including registering what actions are taken during a user session with regards to a story. The state entity records changes in the state of a story for a specific user. The state diagram in **Figure 6.7** shows the states a story can have and the possible transitions between them. There are four states in our system (recommended, read, to-be-read and rated), all of

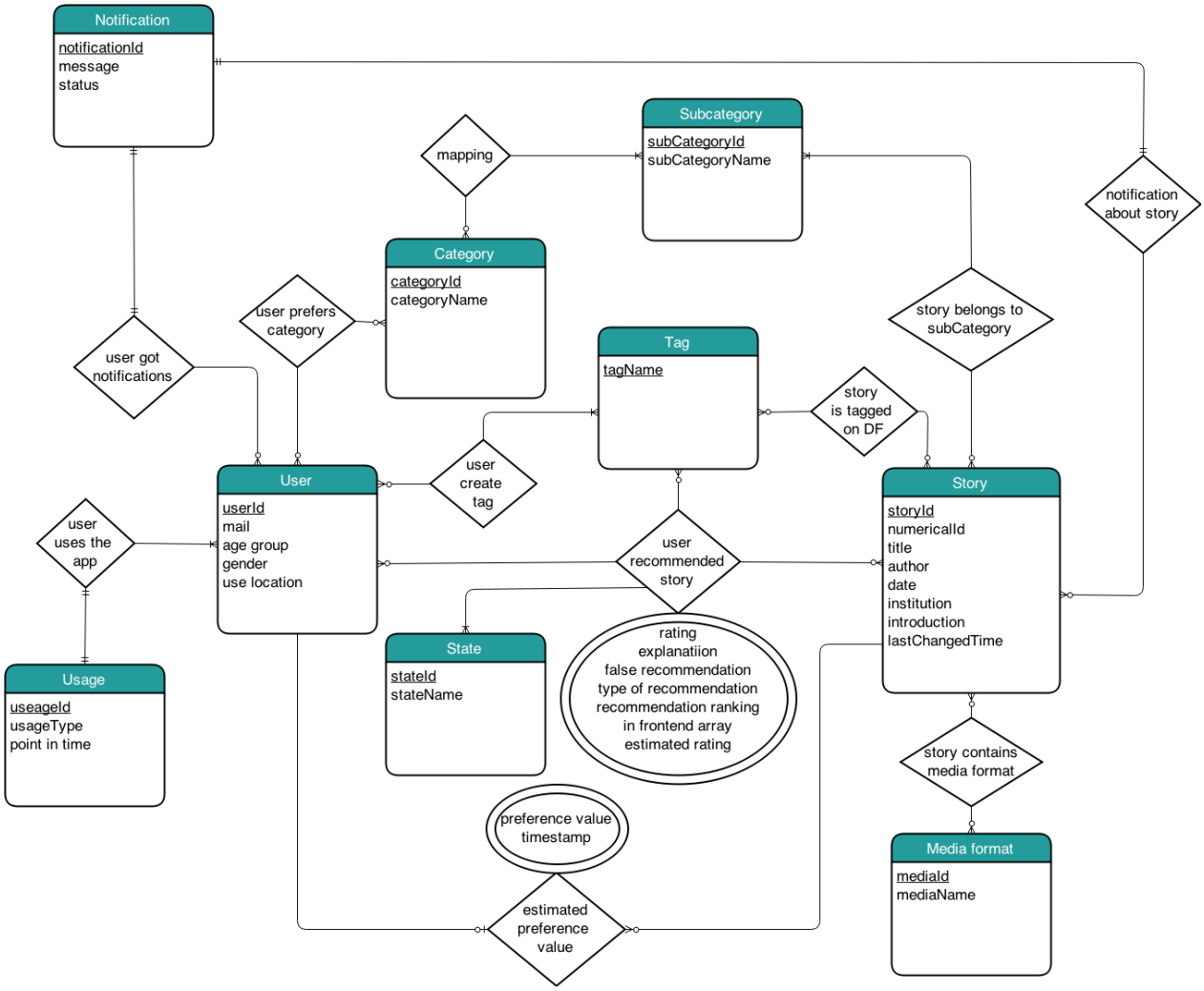


Figure 6.6: ER-diagram showing the data model

which can be recorded multiple times for a story for a given user. The labels on the arrows between states are the events that trigger the transition from one state to another, with one exception: the ”Story not rated”-label (marked with a border around the text). This is a boolean condition which must evaluate to true to enable the transition. For our system, this means that a story can be recommended again after it has been opened (read) or bookmarked with ”Les senere” (to-be-read), but only if the story has not been rated at any point. Another point to emphasize is that a story has no state in the system before it has been ”Viewed” (seen in the state diagram as the initial transition from the solid circle which represent the start point). A story is ”Viewed” when the user sees the story in the recommendation view at front-end. This means that the only way a user can experience a story is to be recommended the story.

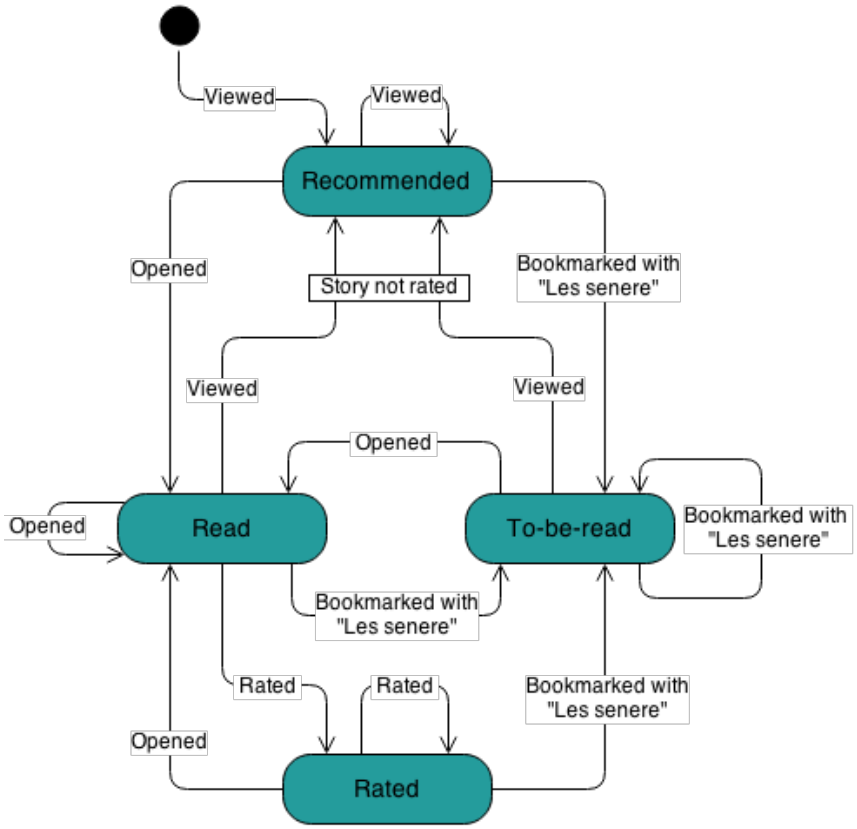


Figure 6.7: State diagram showing states and transitions for a story for one user

6.10 Docker

Docker [**?**] was made to help automation of application deployment. This happens by providing a virtual operating-system-level abstraction. This means that on a server, it is possible to run several virtual operating systems called docker images, which can easily be deployed to another server. This is beneficial for software development, because it means it is easy to setup identical back-ends at different locations. The customer used this on their servers, which made using it during development a good choice as well. A benefit of using docker is that it can directly access repositories on Git. This means that the latest revision is guaranteed to run when starting the back-end. However, there are also drawbacks. It is not easy to update files within the docker image currently running without rebuilding it. This means that to update a single line of code, the whole image needs to be rebuilt. Which means that while the newest revision is guaranteed to be running, any changes made to the application after the start of that docker image requires manually stopping, rebuilding, and restarting the docker image.

A docker image based on Ubuntu was used. In the Dockerfile, which is used to automate the deployment, all required dependencies were installed through Ubuntu's command line. A Linux, Apache, MySQL, and PHP (LAMP) stack and Java were the most critical dependencies. The setup of these tools was written to files which are downloaded from GitHub at runtime along with the back-end source files. After the Dockerfile is run an image is built and launched as wanted.

CHAPTER 7

IMPLEMENTATION

This chapter discusses the details of the system implementation process, both for front-end and back-end implementation, but also how the project was managed throughout the process.

7.1 Project progression

The project progression derives from the product backlog (**Appendix D.2**), meeting minutes (**Appendix F**) and Trello archive and will only briefly summarize what was done at each sprint and only serves as a ”quick tour” through the development process.

Sprint 1 - 23.01.15 - 30.01.15

The first week mainly consisted of getting familiar with each other and the assignment. As soon as the introductions were done, we talked about our personal goals to set the expectation as a group. We formalized the rules of engagement (**Appendix A**) for the group to sign, and also delegated responsibilities. Later we established a meeting agenda with the customer and met with them for the first time.

Sprint 2 - 30.01.15 - 06.02.15

In sprint 2 we began planning and formalizing the requirements. Large sections of work like sketching a WBS (**Figure 5.1**), generating a product backlog (**Appendix D.2**), and making use cases were done this week. Other design work like the architecture and UI mock-ups were also started on.

Sprint 3 - 06.02.15 - 13.02.15

This was the first week of development. We started testing the API (**Subsection 4.1.2**) and finalized some paper prototypes for user testing (**Subsection 7.2.1**) and also completed the design for the database and overall system architecture (**Figure 6.2** and **Figure 6.6**). Research on personalization (**Section 3.4**), the front-end framework Ionic (**Section 3.2.2**), and differences between the two operating systems IOS and Android were noted.

ref cross-platform OS

Sprint 4 - 13.02.15 - 20.02.15

For this sprint we analyzed the existing work done in the past (**Section 3.6.1**) to see if would fit our needs, and continued the research on collaborative filtering (**Section 3.4**). Some group members also spent time learning Docker (**Subsection 6.10**). We mapped the Digitalt fortalt categories to our own (**Section 6.6**) and conducted and evaluated the paper prototype tests as well. During this sprint the database code was also starting to take shape. We did a prioritization of the functional requirements in order to better plan the next sprint.

Sprint 5 - 20.02.15 - 27.02.15

In this sprint we managed to get the harvesting (**Section 6.7**) up and running after the database was ready. A lot of work with the internal models on the back-end was done. The front-end team moved steadily though the initial work on the prioritized views (login, story and list). There was also a revision of the prototype to accommodate the customer's feedback from their user tests.

Sprint 6 - 27.02.15 - 06.03.15

Finalizing the database communication was done in order to fully test the front-end and to get the back-end team started on the personalization. The test plan (**Section 8**) was in the stage of being attuned and formalized for the documentation. After the new prototype was finished, another round of user tests were conducted and the work continued on each of the application views.

Sprint 7 - 06.03.15 - 13.03.15

Some work to meet a report delivery was done early in the sprint. Later in the sprint, the back-end was showing good progress on finding/testing a suitable filtering framework, while the front-end refined the appearance of each view to better fit the user needs. The was also work done on resolving some issues with the server communication.

Sprint 8 - 13.03.15 - 20.03.15

The application was taking shape on many fronts, so this week we worked more on video support and started with the on-boarding section of the application. On the back-end the Mahout (**Section 6.8**) framework setup was on the way. There was also some work spent on testing.

Sprint 9 - 20.03.15 - 27.03.15

This week we polished some features and functionality to hit the milestone Beta

version (**Section 5.5**) in order to prepare the application for some testing in the wild during the upcoming Easter. Further development on the Mahout framework, content-based filtering and fixing of some unforeseen issues on the harvester was also done.

Easter - 27.03.15 - 07.04.15

The application was tested under informal conditions on family and friends, a so-called ”in the wild” test.

Sprint 10 - 08.04.15 - 17.04.15

We dealt with the feedback from usability tests conducted during the Easter. Some platform-specific issues needed some focus to be resolved. We did some more polish on the views and cleared out some front-end bugs. on the back-end, work continued on filtering both collaborative and content-based, which it would for the rest of the sprints.

Sprint 11 - 17.04.15 - 24.04.15

The application is beginning to be nearly finished on a functional level. Resources were dedicated to polishing, and bug fixing became the focus while a lot of testing was conducted on the server. The last big task that the back-end worked on was to finalize the filtering.

Sprint 12 - 24.04.15 - 01.05.15

Sprint 12 was similar to sprint 11, but there was a significant increase in the workload this week due to the upcoming final milestone of having an application ready for delivery to the customer. We intended to accomplish this by the end of this week. This did not happen, but we had aimed for a final build earlier than necessary. We planned for some last minute changes and allowed time to fix undiscovered bugs, which was exactly what happened. We also used some extra time for unit and system testing.

Sprint 13 - 01.05.15 - 08.05.15

This week was spent on various bug fixing for some members for the group, while the majority did mainly report and documentation work. We planned an acceptance test with the customer the following week. We went through the application and did extensive functional testing prior to the customer meeting.

Sprint 14 - 08.05.15 - 15.05.15

During the testing done the week before, more undiscovered issues were uncovered and we needed some more time for polishing and fixing. We had not yet locked in a date for the customer to do an acceptance test, except before the end of the month. The customer agreed to set up a final meeting the following Monday, which we felt would give us time to take care of our main issues, and even give us some time to go the extra mile.

Past 15.05.15

At the start of the week we met with the customer to go through the functionality in conjunction with the functional requirements to see if the customer also thought we hit our goal (See **Section 8.4**).

Work was conducted after the last sprint(sprint 14) this was mainly fixing some bugs, finalizing the report and the application and source code handover. The final burn down of the work progression for all the sprints is shown below in **Figure 7.1**.

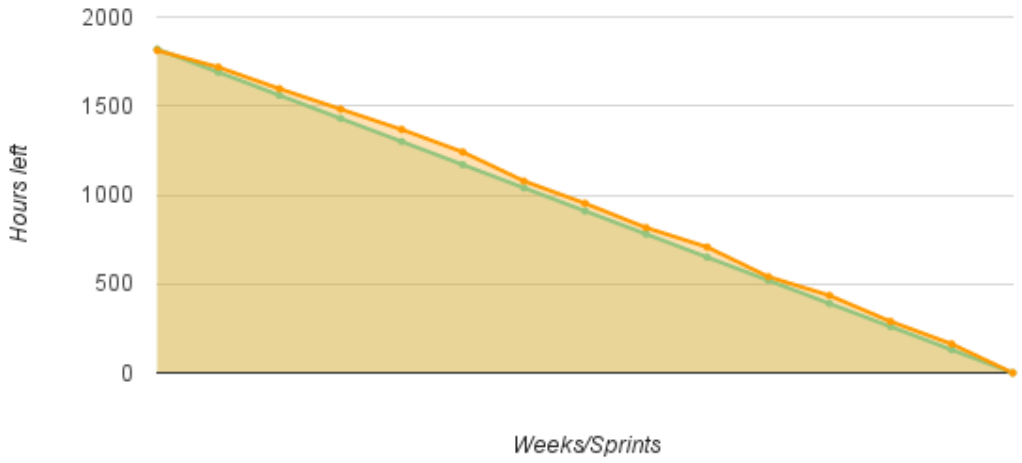


Figure 7.1: Final burn down chart

7.2 Front-end

This section will be an elaboration of some challenges and limitations that arose during the creation of the user interface, both concerning the design and the implementation aspects of the process.

7.2.1 Designing the user interface

The user interface design was an essential part of the project, as the customer prioritized usability over all other non-functional requirements. The design therefore went through many iterations of working on a prototype and continually getting feedback from customer and user tests. This feedback loop was important as the customer did not have all the specific requirements to the application from the beginning, and because making everything easy to understand for the user was also challenging. Balsamiq was first used to create a basic wireframe, but as its functionality was limited, the next iteration of the design was made using Proto.io. This made it possible to receive better feedback on the flow of the app and not just the views individually.

While designing the user interface, some attention was paid to the projects described in **Section 3.6**. The group analyzed this previous work as a basis to decide what would be good or bad design choices for "Vettu hva?". A few general conclusions described in **Section 3.6** were that having static images occupying screen space is poor design, and that lists are normally easier to navigate than maps.

The target audience for the application included both those who have an interest in cultural activities, and also those who do not have any interest or experience about this, so as to encourage more of the general population to discover an interest in the subject. A specific target audience of 16-19 year old teenagers were promoted as a possible focus, because of the possibility of encouraging a young audience to develop an interest in cultural heritage. However, this was not stated until around midway in the project timeline. This fact made it difficult to consider target audience when making design decisions.

The prototype has been through multiple iterations. Early on, it was imagined to have a sort of "magic" discovery function where a user would for example rub a crystal ball and receive a recommended story. This idea was later discarded because the team and

customer realized it would be better usability to present the user with multiple recommended stories that they could simply browse through instead.

Another of the early ideas was for the user to receive a "daily story" or some sort of schedule for being presented with recommended stories. However, due to workload and time constraints, this requirement was heavily down-prioritized. The most important parts were the personalization and usability aspects, so receiving notifications seemed like an unnecessary extra feature.

In the first prototype a media preference view was also included when creating a new user. In this view the user should put the media formats (text, images, video, audio) in order by which format the user preferred. This was discarded in the second prototype, as it was difficult for users to understand the purpose of it and the customer agreed it was not necessary.

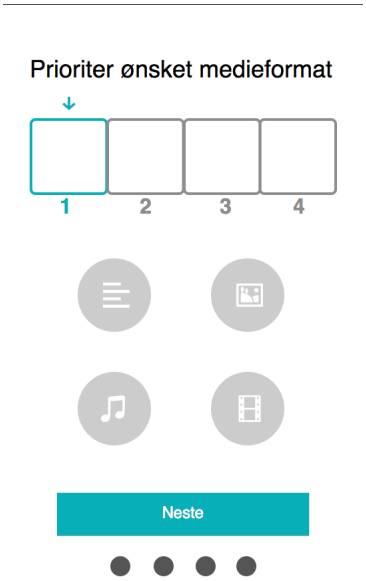


Figure 7.2: Media preference view from the first prototype which was later discarded.

The application uses many different icons in various parts of the interface, and these have been the source of much debate and redesign. The icons used to represent categories were not always understood by users, and some categories like "local tradition and food" were difficult to represent universally with just a single icon. Also the bookmark icon shown in the upper right area of **Figure 7.3** was confusing to some users, and there was a concern

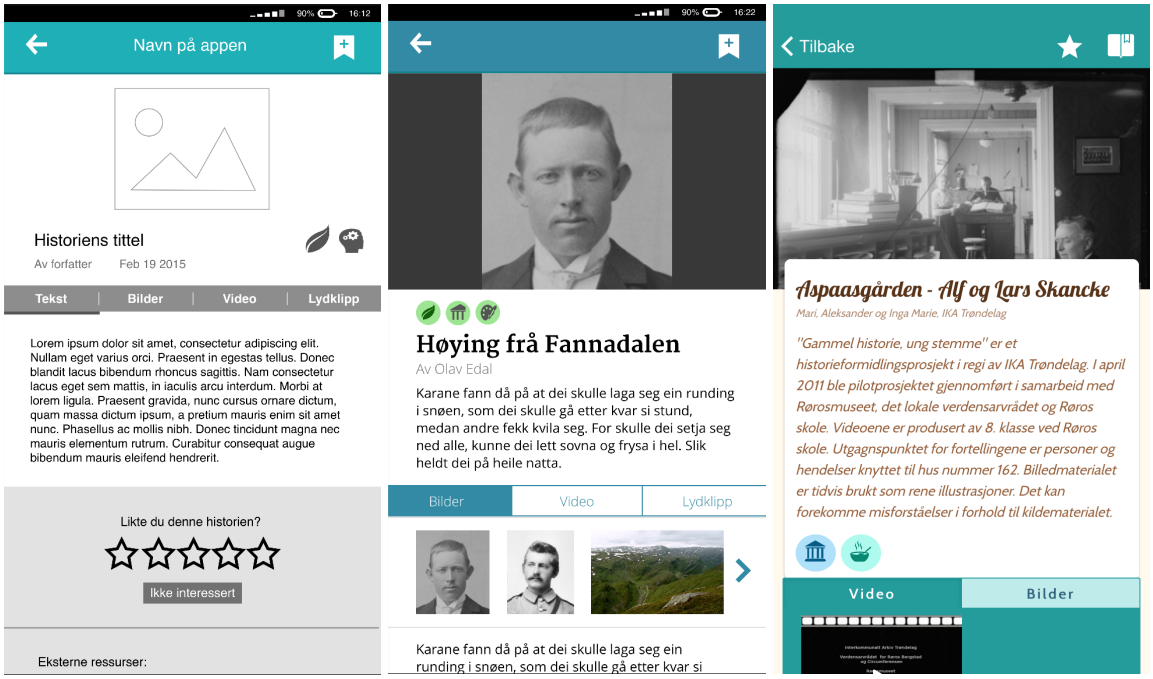


Figure 7.3: Comparison of the story view in the first and second versions of the prototype and the final design implemented.

in the team that this icon might not accurately represent that it allows the user to save the story in a bookmark list.

A big issue for the interface design has been the handling of the different media elements (text, pictures, audio, video) and how these should be positioned relative to each other. For a while the team designed the application to have one tab for each of these four elements in the story view, as shown to the left in **Figure 7.3**. The customer had a concern that this might not be the optimal solution, as a user would for example not be able to read text and view pictures simultaneously. After some discussion, the interface was redesigned so that the text would be persistent, and instead the user could tab between pictures, audio, and video. The resulting design can be seen to the right in **Figure 7.3**.

7.2.2 Implementing the user interface

Early on in development, the team discovered several limitations to the Ionic framework. For example, when using a list to display stories, it was not possible to swipe the list both left and right. The idea was to swipe one way to add a story to be read later, and swipe

the other way to reject a story from the list entirely. Because this proved to be impossible, the views were redesigned into a different solution which was much less based around swiping.

As this is an application for mobile devices, it had to be adapted to work on different screen sizes. The team found that it would most likely be best to target a relatively small screen size and then simply enlarge it for bigger screens. This eliminated the issue of having to compress the components to fit smaller screen sizes and potentially be forced to redesign the whole view to fit small screens.

Adopting accurate naming conventions for the different components has also been a considerable issue. Stories can be saved in bookmark lists, but these lists have interchangeably been called collections and tags in the system. Also when asking a user to input their preferred categories to receive stories from, there has been some confusion because of interchangeably calling these categories for interests, preferences, and categories.

Implementing media, and especially video, has been a challenge in the project. An issue with this has been that playing videos is handled differently on iOS and Android, which had resulted in some bugs that only appear on one platform and not the other. These types of issues have been problematic to fix and has taken up much time. In addition, the videos provided by the Digitalt fortalt website come from different sources. Some of them are Youtube videos, others are Vimeo videos, and there are also other variations. Integrating all these different formats smoothly into the application has been a considerable challenge as well.

Being able to reject a recommended story was a feature that was supposed to be implemented, but it was more difficult than it seemed. The problem was with the slide component from the Ionic framework, which did not do well with dynamic content. When removing a slide from the middle of the list of recommendations, it would not update properly and swiping did no longer work.

It was also decided to remove the ”not interested” option when rating a story. This was considered when testing the prototype, but was not decided until after the development had started. It was not intuitive what the button would actually do, and the user can convey the same ”not interested” message in other ways. In order to rate, the user has to

at least look at the story. If the user does not like the story after looking at it, then it is both more intuitive and useful that the user gives a low rating rather than ”not interested”.

7.3 Back-end

This section describes the development of each part of the back-end. It aims to give a timeline of the development and explain how and why important decisions were made. The different parts described here are Docker, the database, the personalization, the choice of language and the e-mail part of the application.

7.3.1 Docker

Using Docker as a deployment tool proved easier than initially expected. This is mainly because there exists a good amount of documentation and examples online. Initially the Docker setup was based entirely on a project called tutum [**?**]. Using this projects Dockerfile it is possible to only specify the Git repository of an application to have an Apache server running with this application. Furthermore, it is possible to define a password to use for the database. This suited the back-end work very well since it filled the initial requirements. However, as requirements developed and understanding of Docker increased, a more tailored and advanced setup was used. In this setup a second image is used to save the database between restarts, which means that it is possible to update code without having to start the database anew. In addition to this, harvesting from Digitalt fortalt is performed at 03:05 server time, each night.

Once work on personalization started it was necessary to have Java in the Docker image as well. This proved uncomplicated and worked as intended. Passwords and other important keys were put in a separate file that can be included along with the Dockerfile where this is required.

7.3.2 Database

Based on the first version of the functional requirements for the application, an initial ER-diagram was made in the middle of February. At this stage the customer and the team had not come to an agreement on a prioritization of the non-functional requirements. This meant that it was for instance not clear how important the performance of the system would be for the customer, an attribute of the system which would influence how

much info should be stored for each story in the database versus how much info should be retrieved from Digitalt fortalt every time a user views a story. However, the changes to the initial diagram have been relatively minor. Some of the alterations were based on updated requirements from the customer (for instance regarding research data), while others stemmed from the group and were optimization of the data model or changes made to facilitate the personalization.

The relational database was created from the ER-diagram using the mapping algorithm in [**?** , p.270-278]. Some of the tables in the database has the potential for NULL values, but this was accepted as it was considered more important to keep the mapping from one entity to one table. In addition, the database already had quite a few tables and further splitting would make the extraction of information more costly, with numerous join operations.

The team did not understand how to do the personalization until mid March. The decision on how to do this introduced changes to existing tables and the need to create additional tables and views in the database. Mahout had requirements for the input data, which meant that a view was created to store all the necessary data for collaborative filtering. Using this view made it straightforward to put the desired data from the database into Mahouts data model.

7.3.3 Personalization

At the start of the project much time was devoted to understanding content-based and collaborative filtering and how the theoretical descriptions of these techniques could be turned into a practical implementation in our application. When the back-end part of the group had a better understanding on how this could be done, an important decision was whether to use open-source recommender engines or to implement the algorithms ourselves. After studying the math and complexity involved in writing our own implementation it became clear that the best solution would be to use existing recommender engines, even if this would require some adjustments to already written code.

As the project was nearly halfway through when the decision to use an open-source recommender engine was made, the team did not find the time to do a thorough review of the many different alternatives. This increased the probability of choosing the wrong

tool to work with and violated the stated preventive action in the risk list (see **Appendix B**). However, the customer suggested three different recommender engines that might be worth looking into, one of which was Mahout (see **Section 3.5** for an evaluation of the three different engines). In choosing a tool the customer had done some research on, the group found that the risk of making a wrong choice decreased somewhat. In addition, Mahout presented its features in a way that resembled the groups research on filtering algorithms and it was therefore reason to believe that getting started with Mahout would not require much additional research. Of particular importance was the fact that Mahout offered a short explanation on how to do content-based filtering using their recommender engine. Most such engines provided functionality for doing collaborative filtering, but it was not always clear how content-based filtering could be done.

Mahouts website provided some guiding on how to build a recommender which made it possible to use the tool without thorough knowledge of how the different methods worked. Most of the work in the beginning of doing the personalization thus centered around how to gather and produce data in a format acceptable to Mahout and how to treat the output recommendations. Content-based filtering was implemented first, as this was the customer's wish and because the first recommendations presented to a user would always be content-based. Since Mahout does not provide methods for finding similarities between items based on their attributes, the group had to implement this. There exist a vast number of functions to compute similarity between objects. The group did some research on different similarity measures, but the literature did not provide a clear-cut answer to which measure was best fitted to our data. Cosine similarity was chosen for its simplicity and because a story's subcategories lent itself well to be represented as a vector.

The customer requested the use of both item-based and user-based collaborative filtering. There was some doubt in the group as to whether the produced recommendations from the two methods could be combined, but it was found that this was possible as both take the same data model as input. When doing user-based collaborative filtering, a threshold value had to be set to tell the recommender engine which users should affect the recommendations. This concerns the similarity found between users. It was difficult to set this value as the computation of the similarity is done by Mahout and because the group did not know what threshold value would be reasonable to set. The value is set by means of trial and error, looking at the resulting recommendations produced using different values, and by looking at example use of this method.

A challenge in the project has been to assess the quality of the recommendations. Reasons for this has been the limited amount of harvested stories and the use of Mahout as a black box. The first point was a limitation set by the customer. They wanted to limit the amount of harvested stories to ease the evaluation of ”Vettu hva?”. With fewer stories the probability of different users reading the same stories would increase, and thus require fewer users to get the collaborative filtering running (the cold start problem described in **Section 3.4.3** would be resolved faster).

The second point was down to the implementation of the project and mainly concerns collaborative filtering. Content-based filtering has fewer variables and it was possible to check if input categories produce recommended stories that in fact are connected to those categories. Collaborative filtering involves more variables and it was not obvious from the raw data to see which users should influence recommendations for a user and what weights to assign to the different users preference values. In general, collaborative filtering finds patterns in the data that are less obvious found there than content-based filtering. By using Mahout as a black box this pattern finding has been hidden to the team. This could have been remedied by studying the implementation of Mahout, but the time required to do this was not found to be within the confines of the project. A broader test of the application with real users could also have given some insight into the quality of the recommendations, but this was outside the scope of the project.

7.3.4 Language

PHP

PHP is a server-side scripting language produced by The PHP Group that is especially suited for web development [**?**]. It is a general-purpose scripting language often used to provide dynamic content from a web server to a client. During the pre-study period several reasons for choosing PHP as the main back-end language were presented. Firstly, all back-end team members were experienced in the use of PHP. Secondly, as inexperienced users of Docker, more documentation existed on how to implement PHP with Docker than with the other language alternatives considered. Lastly, it was easily combined with the HTTP protocol used by front-end to request and receive data (see **Section 6.4**).

Java
Java is a very popular general-purpose object oriented programming language developed by Sun Microsystems and later Oracle Corporation [**?**]. Java was later in the project chosen as a secondary back-end language. This was because Mahout, which was used to implement the recommender engine (see **Section 6.8**), is written in Java.

7.3.5 E-mail

To create a user in "Vettu hva?" the user only needs to provide an e-mail address. E-mail was chosen as the identifier because it is unique for a specific person. When a user tries to log in and the e-mail does not exist in the database, a new user is created and the user is assigned a user ID and the ID is returned to front-end. A message is also sent to the provided e-mail, to confirm that the user is registered. If the e-mail already exists, the previously created user ID is returned to front-end and the user can continue where he/she left of. The mail service used to send the confirmation e-mail is an e-mail sending library for PHP called PHPMailer. Originally, the plan was to use the built in mail() function in PHP. However, in order for this to work a properly configured local SMTP server was required. This proved to be too much work for this assignment, so the solution was to use PHPMailer with a "Vettu hva?" gmail. PHPMailer can be used to send e-mails from existing mail servers, like Google Mail, by providing a username (gmail) and password.

CHAPTER 8

TESTING

The following sections describe the strategies for the testing levels unit test, integration test, system test, costumer acceptance test and usability test. The unit, integration and system tests were done in an iterative manner. After the test suits were performed and the results were documented, the testers mended the issues in the system that appeared during the test. After the mending, the test suit was executed again. This cycle was repeated until there was no issues left in the system and all test cases got the expected result. The issues that were discovered during all the tests are described in every testing level section.

8.1 Unit testing

The purpose of unit testing is to ensure that every piece of code that is implemented in the system is functional and correct. It was necessary to prioritize which units should be tested, considering the amount of time given to the project. The units in these tests were PHP, Java and JavaScript classes. Therefore it was necessary to use different unit testing tools. Back-end code was tested using the PHPUnit framework [**?**], and JUnit [**?**] and the extension DbUnit [**?**]. The front-end code was tested by using an AngularJS unit testing tool called Karma [**?**]. The classes that were tested are listed under the unit column in **Tables 8.1 to 8.3**. The classes that were left out in the documented tests were tested manually by the developers.

8.1.1 Roles and responsibilities

To get a structured testing experience, the team had to delegate responsibility for the units. The roles corresponded with the roles that were given at the start of this project (see **Section 5.3**), so the testers would have good knowledge of the code and know how it worked. The delegated responsibilities are presented in **Tables 8.1 to 8.3**.

Table 8.1: Shows the delegated responsibilities in testing the back-end part of the system written in PHP code.

PHPUnit testing		
Unit ID	**Unit**	**Responsible**
UN1	Database Helper	Kjersti
UN2	Database Story	Kjersti
UN3	Database User	Eivind
UN4	User Model	Eivind
UN5	Compute Preference Value	Kjersti

Table 8.2: Shows the delegated responsibilities for the back-end part of the system written in Java code.

JUnit and DbUnit testing		
Unit ID	**Unit**	**Responsible**
UN6	Recommendation	Audun
UN7	Database connection with Java	Audun

Table 8.3: Shows the delegated responsibilities for testing the front-end.

AngularJS Karma testing		
Unit ID	**Unit**	**Responsible**
UN8	UI Login	Ragnhild
UN9	UI Story View	Ragnhild
UN10	UI Settings	Roar

8.1.2 Test cases

The testers created test cases and used these as a guide for performing the tests. Each test case has an ID and describes exactly what the test should do, what input data to use and

what is expected to happen when the test is running. An example of a unit test can be found in **Table 8.4**. All the unit test cases are presented in **Appendix E.1**.

Table 8.4: Unit test case for updating rating value

Test case ID	**Description**	**Input data**	**Expected results**	**Result**
Unit 1: Database Helper				
UN1.1	Update one rating value in the database	Random userId, random storyId, updateValue: 2	The function should return true when running the database request.	Pass

8.1.3 Detected and mended issues

This subsection includes some of the most important issues that were discovered during the unit testing, and how they were solved.

In the database communication there were some changes done during the development which made it easier to handle the code. One example of this was to make the database connection return an array where the array keys matched the column names. This way made it easier to access the different values returned from the database. Another issue was discovered with the media format on each story, which was not stored correctly in the database because of an error during the harvesting of the stories from Digitalt fortalt. These types of issues were solves with several checks to see if the information was in the right format or was not null.

During the tests of the first version of the recommender it was running at a time of 11-12 seconds. This was improved by increasing the efficiency of the communication with the database. The recommender was more efficient when the insertion of preference values and recommendations was done with one insert statement to the database instead of 167 and 10 statements respectively. Previously the recommender fetched a whole table from the database with the preference values. Fetching preference values for a specific user instead of all users helped the recommendation module to run faster as well.

8.2 Integration testing

Integration testing was performed after unit testing, and before system testing. After the back-end and front-end code was tested and integrated, the integration testing could be started, and focused on ensuring that these two modules communicated correctly and that data was moving between them in the right manner. Because of the time limitations and the difficulty with learning a new interface to perform integration testing, the developers decided to perform the integration testing with unit testing framework, more particulary with PHPUnit. This testing framework was already known to the developers and therefore easier and less time consuming to use. All the tests were executed by simulating HTTP requests from the UI and checking that the back-end gave the correct response to the specific HTTP request.

8.2.1 Test cases

The test cases were made by first having a closer look at the different components of the system and the data flows between them (See **Figure 6.2**) The integration was performed on the communication between front-end and back-end code, and included test cases made to test all the HTTP request front-end sent to back-end.(About HTTP request in **Section 6.4**). An example of an integration test can be found in **Table 8.5**. All the integration test cases are described in **Appendix E.2**.

Table 8.5: Integration test case for creating a user

Test case ID	**Description**	**Input data**	**Expected results**	**Result**
I.1	Simulate a HTTP request which will log in a user for the first time with an email adress Call getUserFromEmail to the database	email: 'testnr16@example.com, requestType: addUser	The HTTP request should return successfull message included the newly created userId. The database function should return a row with the userId, mail, age_group, gender and use_of_location	Pass

8.2.2 Detected and mended issues

The following paragraph includes some of the most important issues that were discovered during the integration testing, and how they were solved.

The issue that was considered to be the most time consuming one was to update a user in the database. This issue was detected in the test case I.3. When a user was updated in the UI in the application, the unchanged information that should still be there, was deleted. This was attempted fixed several times with methods that were not adequate. The problem was eventually solved by fetching all information about a user in the database, update the fields in the user model that had been changed, and then insert all the information in the database again.

Other minor issues that were handled were syntax-issues, redundant code that created unnecessary confusion, and missing table attributes in the database. Changes in the code were made to obtain better structure in the code, such as dividing long code files into smaller ones and to make sure functions return a response if an error occurs. Due to a misunderstanding between front-end and back-end developers, the database returned names of category preference and story categories, and not ID's, this was also mended after the integration testing.

8.3 System testing

The system testing was performed after the unit tests and integration tests. The test gave the developers a measure of whether the system met all the goals set for the project. The system test included performing a black box test of the system, where the test cases were based on the specified requirements (See **Section 2.1.1**) and the use cases (See **Section 2.1.2**) defined earlier in this report.

In this test, one of the developers was executing the test. Because it is a black box test, the tester executed the test cases with no access to the code. The tester went through all of the test cases one by one and performed the test cases manually. Due to the time limits of this project, the team was not able to write scripts to perform the test cases.

When the system test was performed, the testers evaluated the test results and then de-

cided if the system as a whole fulfilled all goals for the project. If issues were detected, the developers would mend them and run the tests again. This cycle was repeated until the all the test cases got the expected result. The test should, if done in the expected manner, help the developers of this project to verify and validate if the application meets all the requirements.

8.3.1 Test cases

Each test case has a test identifier and an approach for the tester, and a description of what is intended to happen when the test case is performed. The tester will be referred to as "the user". Some of the test cases have a dependability on other tests. If an issue is detected in one test case, it might cause issues in its dependent test cases. **Table 8.6** presents an example of the test cases that were used. The whole system test case document is in **Appendix E.3**.

Table 8.6: System test case for creating a recoverable profile.

Test ID	T1
Test Item	Create recoverable profile
Approach	The user locate and press the "register user" button in the app. Applies the email in the correct format.. The response is valid and the user gets feedback.
Input data	"newuser@example.com"
Expected results	The user writes the correct email address and get the correct feedback from the system: "Kontakter server" and will be directed to the startup page.
Testing task	1. Click "create user"-button. 2. Apply email address to the email input field 3. Receive feedback from the system 4. Check email inbox to se if the correct mail from the system was received
Depends on tests	
Pass/Fail	Passed

8.3.2 Detected and mended issues

During the system testing, the testers found repeating recommended stories. This was solved by checking the front-end array and the top ten recommendations and the ratings done by the user for duplicates.

One category icon looked completely different in different views. The reason for this was because this icon had been updated in one part of the code, but not everywhere. This was quickly found and solved.

Sound clips were not working on some Android versions. This was a tough issue which was eventually fixed by using dedicated plug-ins for media, and rewriting the functions for retrieving audio files.

An issue was detected when logging out from a user and then logging in with a different user. The current user would then get the same recommendations as the previous user. The issue was that a user's ID was not cleared properly from the cache when logging out.

Sometimes the recommended stories would take a long time to load. This was fixed by optimizing the harvester.

Can not scroll down if you are scrolling by touching the frame of video, picture and sound.

It was discovered that there was no way to delete the user-made lists. This was easily remedied by adding an ”X”-button on these lists to allow deleting them.

When a user created many lists, there was no way to scroll through them. This was remedied by adding a scroll function to the menu.

8.4 Customer acceptance test

The customer acceptance test (CAT) was executed during the whole software development life cycle. After a sprint, the customer tested the product, evaluated and gave feedback. In the early stages of the project process this included testing of the prototypes. When

working software was delivered to the customer after a sprint, the customer used their own real input data to test the behavior of the system. This kind of testing might reveal a different result than from a regular unit or system testing, when the data could be more realistic since the customer defined it. The customer brought feedback either in meetings or through email. The planned delivery dates are presented in **Section 5.5**.

Table 8.7: Customer acceptance test - First paper prototype

Delivery	First paper prototype
Date	20.02.15
Comments	Intuitive interface. The selection of categories is good and fast. Category icon in the listview looks very good.
Issues	There should be a description of why the user has to sign in by email, and give the user the option to choose age group and gender. The customer thinks it is easier to click something than to drag an icon from one place to another. The customer prefers one story per view when the user is browsing recommended stories. The swiping from one story to another should be explained. The customer wants a to-be-rated list and a to-read list. The trash can icon is confusing. It is okay to not prioritize the notifications in the app.

Table 8.8: Customer acceptance test - Second prototype presented in prototyping tool

Delivery	Second prototype presented in prototyping tool
Date	27.02.15

Comments	The customer is overall pleased with the prototype, but they have some constructive comments. There are some confusing icons, some lack of consistent terminology, missing introduction for the app, suggestions for different text for buttons and headlines.
Issues	Overlap between not interested and one star rating. Remove the not interested. The author of a story expects that the story is presented the way he/she made it. It would be more correct to have the elements of the story together, in accordance with the authors intention. The elements of every story are now separated with the tabs in the storyview.
Suggestions	Have a number connected to the rating stars. It is interesting for the customer to know if the user prefers picture, video or sound. The system should log this for every user. It is important to collect the information about the user of the system(age, gender, preferences). It should be discouraged for the user to skip this step. Profile information such as age and gender can not be changed after the specification is once set by the user. Have a little text that appears when you hover over the category icons, or apply a function where you can press a button and reveal the descriptions of the categories. Have the option to share the saved stories on social media. The customer has given this a low priority.

Table 8.9: Customer acceptance test - First working software

Delivery	First working software
Date	17.03.15
Comments	The customer likes the user interface of this version and says that it is not necessary to add more functionality, except for the concept view that is not yet implemented. The customer thinks that fetching the stories from Digitalt fortalt is working fast enough.

Issues	Missing a concept description for the application. There are some stories that do not have categories connected to them, these stories should be included in the collaborative filtering.

Table 8.10: Customer acceptance test - Second working software

Delivery	Second working software
Date	20.04.15
Comments	The customer is satisfied with the appearance and structure of the story view.
Issues	There are only 3 stories in the recommendation view. When you choose interests in the setup of the application, the interests are not marked clearly when you click on them. When choosing a gender in the setup of the application, the selection is not stored in the settings view. The customer discovered that some stories have mismatched icons connected to them. Stories that include a sound clip, do not show this immediately. The sound clip is located inside a tab, and the user would have to click on this to reveal it. Some stories include video clips that are not playing. A user has to sign in every time to visit the app, the user is not remembered. A story that is read is not automatically stored in the 'Read List'. Uncertain about the cross in the corner of a story in 'recommended stories view'. The use of this button could mean two things: Either that the user is not interested in this story, or that the user has read this story and just wants to close it for now. The application does not at any time give the user new recommended stories. Missing some kind of feedback when a user has changed the interests in settings. The system should let the user know that it is trying to get new recommended stories based on the new interests. Some stories do not have a picture attached to them. The customer is suggesting a default picture for these stories.

Table 8.11: Customer acceptance test - Final product

Delivery	Final product
Date	01.05.15
Comments	Customer reported positive feedback and expressed contentment towards the final product. The last meeting was spent on going through the requirements list found in **Appendix C** to see if the product met all requirements. There were some comments during the review. R18 is not completed but the customer sees no problem with this and is satisfied with how the system looks now. Regarding R22, The information about the app is now hidden in the settings view and should be moved to a different location in the menu. R12 is mostly achieved, with the exception of not being able to reject stories from the recommendation view. This functionality was removed because it caused bugs in the application. Apart from these minor issues, the customer said that they were very pleased with what we have accomplished.
Issues	

8.5 Usability testing

The user testing was performed by the front-end developers. The preliminary work for the user test included doing an analysis of the requirements, and using these as a base for making several test cases. A test case included several test steps that the user followed, and the tester observed how the user reacted in every test case. After each test finished there was a discussion with follow-up questions the user had to answer, for the developers to get a better insight into what was problematic and what was easily understandable. The tests were conducted by both the customer and the team separately, and the customer involved three expert users and one external user in their tests.

The user testing was performed over several days where the first test was conducted on the 21st and 22nd of February 2015. This was an early paper prototype while the second

session was with the revised prototype on digital devices[**?**], this was carried out on Feb. 28th and March 1st. All the tests were performed in accordance with the guidelines and tips provided in [**?**].

8.5.1 Test users

The test users were asked in advance how much and how many applications they use on handheld devices. They were also asked if and how much reading they see themselves generally doing. The **Table 8.12** below describes the users that were used as test subjects for the usability testing.

Low - 1-2 applications/ 1-2 times a day.
Medium - 2-4 applications/ 2-4 times a day.
High - more than 5 applications/ more than 5 times a day.

Table 8.12: Test group

Gender	**Age**	**Application usage**	**Other**
Female	26	Medium	Reads some
Male	32	High	Does not read much
Female	51	Low	Reads a lot
Male	31	High	Reads some

8.5.2 Test cases

On each test, the user received a set of tasks to complete. These tasks were the following:
Create a user

1. Login
2. Set personal data

3. Add at least 2 categories

Read a story

1. Browse the suggestions
2. Add a story to ”Les senere”
3. Read another story and look at images

Find and delete

1. Find the ”Les senere” bookmark list
2. Delete a story from the list
3. Read another story from the list

Change settings

1. Navigate to settings
2. Change preferences

An example of usability test results can be found in **Table 8.13**. All the usability test cases are described in **Appendix E.4**.

Table 8.13: Usability test example

View	Desired next step	Comments
6. Recommendations	6	**Feedback:** The read later button is not understood. Users do not understand that it places the story in a list. Users do not understand that they can swipe in this view, or what they can touch. The users feel a bit ”dumped” into this view with no explanation. **Changes:** Remove read later button and add a bookmark icon at the top right instead. Add a card deck or something else to show users that they can swipe. Add a loading animation for when the view is retrieving stories. Change the title of the view to ”Anbefalte historier”. Make it clear that the card can be touched and make it possible to remove the card.

8.5.3 Summary

The time set aside for each test was about 20 minutes for the test and about 10 minutes for follow-up questions. Observations during the test and the answers provided in the follow-up were implemented in the view feedback and/or the general changes.

To summarize the tests; each user had his or her own problems with the application but on the whole the user is able to do the task they are asked to complete. So considering the scenarios and use-cases, the application is translating and making the user understand what its functionalities are. There are of course some major inadequate parts, but this is as intended for the test and we are aware of the missing parts.

When it comes to the intuitive understanding, the application has some work to be done. This is covered on a per view basis. And since the user group is not narrow enough we aim to design for everybody and this will act as a constraint for the UI.The parts where users are confused are mentioned in the views section.

Paths are on the whole followed by the majority of the testers with some deviation, but this is expected since the prototype is a bit unclear on its formulation on some of the views. But to conclude, the users are almost without issues following the scenarios to the point.

CHAPTER 9

EVALUATION

This chapter describes the final evaluation and reflection by the team for different aspects of the project. This includes positive and negative sides, what worked well and what could have been done differently.

9.1 Product quality

We are satisfied with the functional and aesthetic aspects of the application. The customer also expressed that they were happy with the final result. Regarding the requirements described in **Appendix C**, we completed all the requirements except for requirements with a low priority because of time constraints. The last evaluation with the customer of how well we had met the requirements was described in **Table 8.4**. The final application passed all the different tests (unit, integration, system, usability) as shown in **Appendices E.1 to E.4**. The system can still be further developed in the future, but the group has concluded that we are pleased with what has been accomplished on the application during these 4 months of work.

While the number of harvested stories — 169 — sets some limits on the quality of the recommendations from the user's point of view, some things could have been done to improve the recommendations from a system point of view (i.e. to produce as good recommendations as possible with the available data). This includes testing and optimization of different variables such as the weights used to compute preference values, the formula for computing similarities between stories, the use of different Mahout methods in the recommendation process and the use of Mahout methods to evaluate recommendations. The reason why this has not been done is lack of time, since a working application has been prioritized.

Some issues with the user interface remain, related to different versions of the Android OS. For example, on very small screen sizes, the views may get significantly compressed and difficult to read. On Android version 4.0, 4.1 and 4.3 we have discovered a bug in the

recommendation view where the header image takes up the whole size of the story card. A few other small visual bugs also remain, because there are many versions of Android and there was no possibility for us time-wise to perfect the application for every version. On iOS systems and Android version 4.4 and above however, the application should work as intended, which was our main goal.

9.2 Development process

We used the agile development methodology scrum, which has helped our work. The frequent meetings were efficient to coordinate our work and share progress, which was a necessity in a group of 7 members. Scrum is also well suited to respond to requirement changes, which was relevant in this case as the requirements for our application had to be reevaluated and changed several times over the course of the project.

9.3 Project management

Project management in this section details how we planned and followed up our project, how well we complied with the process model, and how effectively we distributed work.

We could have estimated time better, as it was significantly off and some tasks proved bigger or smaller than we initially thought. It caused moments of stress when we spent too long on some tasks and did not delegate enough time, which meant we had to down-prioritize other tasks. However, estimating how much time to use for a task is difficult, and more so when there is little to no experience with said task within the group. A better option might have been to guess units in stead of hours to remove the pressure of completing a task within the time limit.

It was hard to use the scrum sprints effectively. The sprints did not always have concrete endings, and sometimes it was hard to decide what tasks to work on next. Occasionally we ended up planning to research, instead of specific tasks. We could have done more planning in between sprints and specified the tasks more clearly.

The role distribution has worked well. Every person performed well in their tasks and we finished the tasks that we wanted to. By dividing up the tasks to small packages, it was possible for each person to work on a specific task at a time, while not having to worry

about other factors.

The requirements have been subject to many changes, and because of this, scrum has worked quite well for us. The process has allowed us to adapt and change the application as needed to reflect the changes in the requirements.

9.4 Team

We had meetings often and kept each other up to date about status, and maintained good communication in general. We agreed about most things and we set ground rules at the beginning on how to work together. We were able to coordinate with each other so we could work independently and still have minimal problems. Everyone contributed and took initiative to get work done. Each member was quick to point out issues and we took initiative to solve problems as fast as possible.

However, it should be mentioned that we divided up the work tasks, for example into front-end and back-end, and these divisions were permanent. This is not a problem in itself, but it meant that the back-end staff did not know much about how the front-end was implemented, and front-end staff did not know much about how the back-end worked. The level of individuality might at times have been too high because of this.

Sometimes the creativity stagnated in the group, even though it was not necessarily anyone's fault. We had to go back and forth several times to come up with some solutions. Examples of this were coming up with the name of the application, and also figuring out creative ways to make the user interface user-friendly. On the positive side, the members in the group have been able to come up with good solutions when coding, that have worked well in the end. By constructively criticizing and testing continuously, we have been able to reach good solutions for the implementation. It has been easy to bounce ideas off each other and make decisions as a team.

Cooperation inside each subgroup (front-end and back-end) has been very good. The members of front-end worked well together, and the members of the back-end also worked well together. The response from other group members when one person had a trouble has also been very good, and issues has been resolved quickly because of the tight and efficient communication.

Communication between front-end and back-end staff could have been better. One group did not always know much about what the other did until very late in the project. Even though we did have meetings to share progress, when it came to the code specifics, each group was largely unaware of how the other group had implemented it. The team members could also have been better at communicating when they would not be available for meetings, or show up late. Because of this, numerous meetings were unnecessarily delayed because of this.

In our group we had similar competencies, but different levels of experience. It strengthened our group in the way that some members has special experiences that were useful to perform certain tasks. For example Unix experience, as well as database knowledge. And over the course of the project, a great deal of new experience was gained such as in the front-end framework used and the tools this required.

9.5 Customer interaction

The customer meetings have been frequent (one meeting every week) and this has been useful. We had many things to discuss, and issues we had to reach an agreement on with the customer. The communication was sometimes difficult, because we did not always understand what the customer wanted, and they did not always understand our thoughts and ideas on how to solve problems.

It has been useful for us to prepare for the customer meetings by writing an agenda before each meeting, and discuss in the group what we wanted to talk about with the customer. On the other hand, we could have been better at coming to an agreement inside the group about issues before each meeting. Occasionally we went to a meeting and team members gave ideas and opinions that not all the other team members understood or agreed with.

We could also have been better at clearly stating our ideas about our solutions to the customer. There has been several incidents where the customer did not understand or agree with our solutions, which cost us time as we had to reevaluate parts of the design.

9.6 Limitations

Before the start of the project, several constraints were identified in advance, and were described in **Section 3.1**. These factors as well as others limited the group in various ways.

- Time was the most prominent limiting factor. For 7 people with an estimated 20 hours of work each week per person, it was not possible to implement every functionality and idea that we wanted to include. This limitation was handled by using scrum to effectively organize our work, as well as making a prioritized list of requirements to make sure that the most important parts of the application would be completed first.
- The framework limited us in some way. As discussed in **Section 7.2.2** there were some of our early design choices that proved to be very difficult to implement with the chosen framework. This was handled by redesigning the interface in a way that could more easily be implemented with the framework.
- The experience and knowledge in the team was at times a limiting factor. **Table 1.1** shows the team's background competencies and also shows that there was little experience with mobile application development. We also had no experience with recommender engines, which meant that in order to bypass this limitations we had to allocate some time to research the topics that we lacked knowledge in. This limitation therefore led to a further restriction on our time.
- We have received various feedback on the report from different supervisors and other sources as well. Some of these feedbacks contradicted each other, and this limited us to using our own discretion for the choices we made with regards to the report. Some of the feedback was of a subjective nature, such as how to structure and order the different sections of the report. This was the sort of feedback that we occasionally had to disregard because the different parties that reviewed the report each had their own arguments for which structure provides the best possible readability.

9.7 Lessons learned

Throughout the project, we discovered multiple things that we could have done differently. Here are the top three lessons we learned from this subject.

- We have learned that it is important to plan the work for each iteration properly. This ended up being a bit confusing when we did not always know what tasks to work on for each sprint.
- It is important to evaluate the requirements often, and make sure that we have understood them properly. Requirements need to be referred to often while implementing the application, and they need to be assessed and discussed during customer meetings so that they are understood the same way by both the group and the customer.
- We should have researched how to implement the personalization techniques earlier. This was started very late in the process and took up a lot of time. The reason it was delayed to begin with was because we wanted other functionality to be implemented first, but in retrospect it should have been started on earlier than we did. The necessary level of research needed for this was underestimated.

CHAPTER 10

CONCLUSION AND FUTURE OUTLOOK

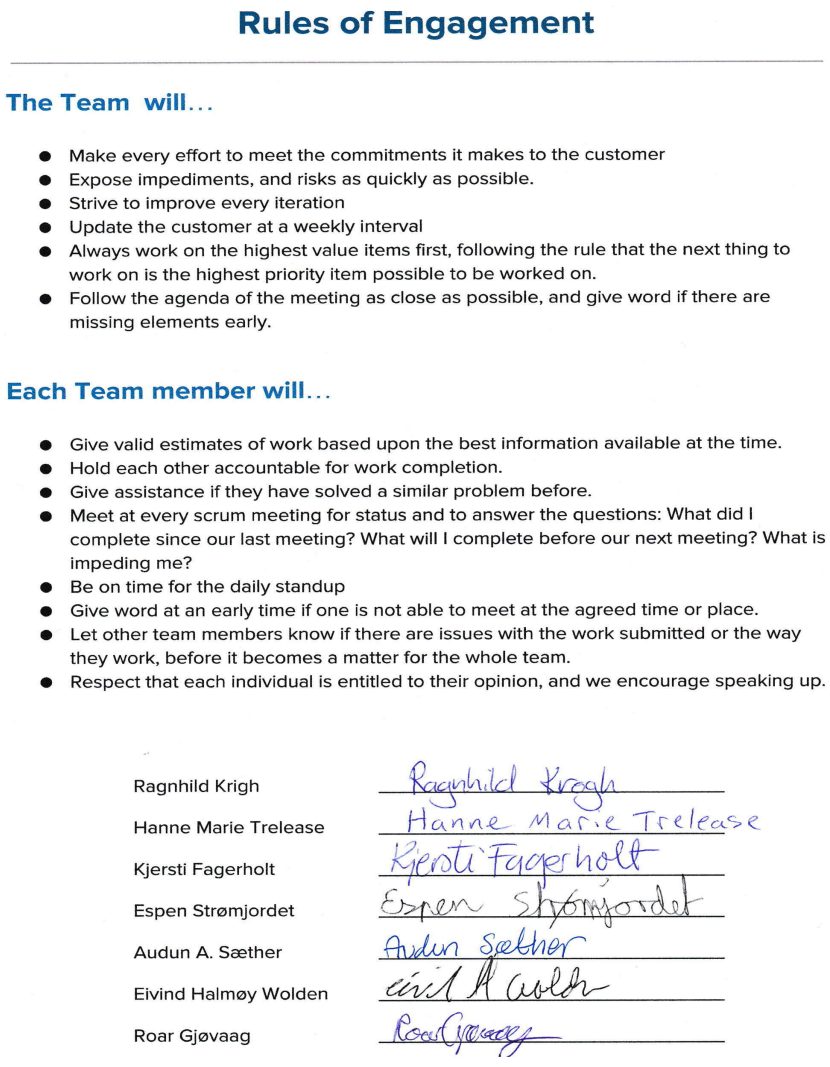
This report has detailed the initial development for the mobile application ”Vettu hva?” and attempted to give the reader an understanding of how this application facilitates the discovery of stories concerning cultural heritage, and also encourages users to develop an interest in this subject.

Personalized content is a central part of many applications and web sites in the present time, and it is a topic that is continuously being researched and improved. Social media and collaborative solutions are increasingly becoming a standard in network-based systems. It is likely that applications such as ”Vettu hva?” will have more elements of collaboration between users, and more interaction with social media in the near future.

Some future plans for ”Vettu hva?” include adding functionality for sharing stories on social media such as Facebook. There also plans for adding filtering based on geolocation. Once the application has gone through several rounds of testing, it will be important to use the data gathered from the tests. A first step would be evaluating the recommendations and looking at how they can be improved. Furthermore, fine tuning the interface would also be a natural progression after further testing. There are many possibilities for future work, and several directions ”Vettu hva?” can take.

APPENDIX A

RULES OF ENGAGEMENT



Rules of Engagement

The Team will...

- Make every effort to meet the commitments it makes to the customer
- Expose impediments, and risks as quickly as possible.
- Strive to improve every iteration
- Update the customer at a weekly interval
- Always work on the highest value items first, following the rule that the next thing to work on is the highest priority item possible to be worked on.
- Follow the agenda of the meeting as close as possible, and give word if there are missing elements early.

Each Team member will...

- Give valid estimates of work based upon the best information available at the time.
- Hold each other accountable for work completion.
- Give assistance if they have solved a similar problem before.
- Meet at every scrum meeting for status and to answer the questions: What did I complete since our last meeting? What will I complete before our next meeting? What is impeding me?
- Be on time for the daily standup
- Give word at an early time if one is not able to meet at the agreed time or place.
- Let other team members know if there are issues with the work submitted or the way they work, before it becomes a matter for the whole team.
- Respect that each individual is entitled to their opinion, and we encourage speaking up.

Ragnhild Krigh — Ragnhild Krigh
Hanne Marie Trelease — Hanne Marie Trelease
Kjersti Fagerholt — Kjersti Fagerholt
Espen Strømjordet — Espen Strømjordet
Audun A. Sæther — Audun Sæther
Eivind Halmøy Wolden — Eivind H Wolden
Roar Gjøvaag — Roar Gjøvaag

Figure A.1: Rules of engagement document

APPENDIX B

RISK LIST

L: Likelihood (1-9)
I1: Impact (1-9)
I2: Importance (Likelihood * Impact)

Table B.1: Risk list

Description	**L**	**I1**	**I2**	**Preventive action**	**Remedial action**
Underestimate the time planned to use for assignments	8	8	64	Estimate a little higher. Continuous meetings. Continuous status update on tasks	Extra work hours and help each other. Have a clear prioritization of tasks so that some less important tasks can be delayed if needed.
The group does not deliver updated information on the report and cannot maximize the quality of the feedback obtained from the supervisor	6	7	42	Always make sure the report meet the demands upon delivery	Ask concrete questions to the supervisor or other competent acquaintances of the group members
An issue in the code that is not understood or can not be fixed.	5	8	40	Comment on the code, and talk with each other about the work done on the code	Get help from the supervisor or other people involved.

Table B.1: Risk list

Description	L	I1	I2	Preventive action	Remedial action
The group does not receive quality feedback from supervisor	6	6	36	Be prepared for supervisor meetings. Prepare concrete questions and discuss issues with supervisor	Ask qualified acquaintances to read and give feedback on the report.
Project complexity / Project too difficult	5	5	25	Do not plan many complicated tasks	Downgrade demands
Poor communication with the customer leading to misunderstandings and doubts about the progress of the project	4	6	24	Be well prepared for meetings by establishing an agenda for the meeting and sharing it with the customer beforehand. Establish a good customer relationship. Send email to the customer for clarification	Send email to the customer for clarification. Cooperate with customer to reprioritize tasks
Poor communication within the group leading to misunderstandings and doubts about the progress of the project	4	6	24	Write meeting minutes to document decisions. Have frequent meetings where every team member explains what they have done and what they are planning to do.	Make a group decision to solve the misunderstanding
Wrong choice of development tools	4	6	24	Do thorough research before deciding which tools to work with	If early in project, consider changing tools

Table B.1: Risk list

Description	L	I1	I2	Preventive action	Remedial action
Absence of group member(s) over a period of time	6	4	24	Every member of the group should be aware of what all members are working on, so that they can step in and take over the absent person's tasks	When the progress is halted by a person's absence, other group members should take over the tasks needed for further progress
The workload is distorted. Some members of the group work too much, while others work too little.	5	4	20	Continuous meetings, after every meeting the team members delegate assignments. The leader of the meeting also have to make sure that everyone gets approximately the same amount of work.	Redelegate work tasks within the group.
Poor choice of programming language. It becomes difficult to produce the product before deadline	2	9	18	Good research and discussion in the group. Do not choose a language that some people in the group do not have experience with	Reevaluate non-functional requirements with the customer
Customer changes requirements	6	3	18	Constant communication with customer	Use an agile development process to better adapt to changes
Data loss	2	8	16	Local copy and regular backups	Restore latest available backup

Table B.1: Risk list

Description	L	I1	I2	Preventive action	Remedial action
Disagreement within the group on key issues in the project	3	5	15	Establish ground rules regarding how to discuss key issues and how to make decisions final. Make democratic decisions	If the disagreement cannot be solved, one may involve the supervisor
Personal conflicts between group members	2	7	14	Establish ground rules for the team so that emerging conflicts are solved as early as possible	First try to solve the conflict between the group members in question. If this does not work, involve the whole group. A last resort would be to involve the supervisor
Product does not meet requirements	2	7	14	Request product feedback from customer	Assess which changes must be made and prioritize the most important parts to change
Overestimate the time planned to use for assignments	3	4	12	Investigate the assignments so it is clear what they include, and how much effort it would take to perform them.	When the assignment is done long before the estimated time, use the extra time on the more time consuming assignments.
Missing deadlines	1	8	8	Have frequent meetings and plan well	Do extra work hours to finish the work as quickly as possible

Table B.1: Risk list

Description	**L**	**I1**	**I2**	**Preventive action**	**Remedial action**
Product is not user-friendly	3	2	6	Perform usability tests	Assess which changes must be made and prioritize the most important parts to change
Technical issues(server down)	2	3	6		Use the backup

APPENDIX C

REQUIREMENTS

Table C.1: The requirements listed up and prioritized by the customers wishes

ID	Name	Description	Priority	Use Case Ref.	Comments
General					
R1	Language	The documentation should be written in English. The application will be written in Norwegian and the stories from Digitalt fortalt will appear as they are, most of them in Norwegian.	H		A non-functional requirement
R2	Cross-platform	The application should run on both Android and iOS.	H		A non-functional requirement
R3	Cross-platform design	The application design should appear similar on both Android and iOS.	H		A non-functional requirement

Sign up /Sign in view					
R4	User recovery	The application should provide the opportunity for the user to enter email address, which then becomes the user identifier in the system from the user's point of view.	H	U1,U2	This means that the user can access the profile from different devices.
R5	Anonymous sign in	The application should provide the option to enter the application without registering a user by mail address.	H	U1	Device remembers user. User got id in database, but not mail
R6	Demo view	It should be possible to run the application mode in a demo view where the system cannot identify the user	L		
R7	Personal info	The application should obtain some personal info about the user, such as age group and gender.	H	U3	Only for research purposes
Preferences/Settings					
R8	Initial specification of preferences	The user should be asked to set preferences in the startup process.	H	U3	
R8A	Set category preference	The user should be asked to enter a number of preferred categories.	H	U3	

R8B	Set location preference	The user should be asked to specify a preferred location.	L	U3	
R8C	Set notification preferences	The user should be asked to set some preferences about notifications	L	U3	
R9	Changing preferences	The application should provide a settings view where the user can change the preferences.	H	U9	
Main view: Browse recommended stories					
R10	Show list of recommended stories	The application should provide the user with a list of recommended stories based on the set preferences. The stories is presented by a picture and a short text harvested from Digitalt fortalt.	H	U4	
R11	Recommend story outside user's preferences	The application should once in a while recommend a story outside the user comfort zone, i.e. a story that the recommender algorithm does not pick out.	M		Purpose: To broaden the user's horizon. The purpose is not to test the algorithm

R12	Make decision about story	The application should provide the user with three options regarding each recommended story: to choose to read the story now, to reject the story or to save the story for later	H	U4	
Story view					
R13	Present story	The application should present the chosen story in a specific story view.The presentation of the story should be in accordance with the presentation on Digitalt fortalt.	H	U6	
R14	Give feedback / rating on story	The application should provide the user with the opportunity to rate the story. The rating is in the form of a star system with 5 stars.	H	U7	
R15	Bookmark story	The user should be given the opportunity to connect a story to bookmarks. The bookmarks could be predefined by the system, like ”Les senere” or defined by the user	M	U5	
R16	Link to Digitalt fortalt	Every story should include a link to the corresponding story on Digitalt fortalt.	H	U6	

R17	Explain why a story was recommended	The application should provide an explanation why a given story was recommended.	H	U4	Could just be a general statement like: ”Other users who liked similar stories to you, also liked this one”
List view					
R18	Show stories connected to a bookmark in a list	The application should show a list of bookmarked stories for each bookmark. The stories is presented by a picture and a short text.	M	U8	
R19	Choose bookmark	The application should provide the user with the opportunity to choose different bookmarks. Bookmarks include: to-read,read and user-defined bookmarks	M	U8	
Notifications					
R20	Notifications outside the application	The application should send a notification to the user’s device at the time specified in the preferences	L		In agreement with the customer this requirement will not be met

R21	Notifications inside the application	The application should create a notification after a defined amount time to remind the user of stories that have been read but not rated	L		In agreement with the customer this requirement will not be met
About app					
R22	About the application	The application should include an about section, which should include basic info about the project. This include references to TAG CLOUD.	H	U10	
Quick tour					
R23A	Quick tour at start up	The application should provide a new user with an explanation of the application at start up.	H		
R23B	Quick tour in menu	The application should provide the opportunity to revisit the quick tour via the menu.	L		
Personalization					
R24	Use content-based filtering	The application should use content-based filtering to recommend stories to user initially. This should in particular be based on category preferences.	H		Implement this before R25.

R25	Use collaborative filtering	The application should collaborative filtering to recommend stories when the user base is large enough for the algorithm to be effective.	H		
Research					
R26	Gather data to SINTEF for research	The application should gather and store information about the use. This include frequency of use, success rate of recommendations and perhaps other things.			Detailed list of what this included provided in mail from the customer. The customer will view this data through the database

APPENDIX D

PROJECT MANAGEMENT

D.1 Status report example

Status report week 6

Introduction
This week has mostly been spent organising and making decisions that will impact the whole project.

Progress summary
The decisions that have been made will decide how the work is distributed in the coming weeks. Work has been made on defining goals and milestones. Furthermore, some of the tools to complete the given tasks have been found.

Open / closed problems
Closed problems:

- A cross-platform framework have been chosen.
- A rough estimate of what needs doing, how long it will take and when it is due has been performed in the form of a product backlog.
- A list of functional requirements have been compiled after a discussion with the customer. Use case diagrams and scenarios have been made.
- Justification on some of the choices made so far have been written for the report:
  - Scrum
  - Framework
- Complete a WBS chart.
- A rules of engagement have been signed, this helps solidify what is expected of every member in the group.

Open problems:
No specific ongoing problem at the end of this week.

Choosing a cross-platform framework was a difficult process for various reasons. There is not much experience in the group using such tools. Additionally there was an internal debate about what is expected from the customer and what does the team expect the end product to look like. This was discussed in light of the the constraints imposed from various aspects. However, the team members now feel confident that an appropriate tool have been chosen and are aware of some limitations this leads to.

Understanding properly what the customer wants and prioritizes has also been a focus this week. While this sounds easy enough, the technical details are often lost in communication. This is something that will require constant feedback and monitoring so that the project stays on track in regards to what is desired by the customer.

Planned work for next period

- Familiarization with the chosen framework.
- Familiarization with digitaltfortalt.no API.
- Creating a design prototype is a goal, this will unify the group and make sure all members are working towards the same goal. Furthermore, this will explore what options there are and highlight any basic flaws in design.

D.2 Product backlog

Task	Time (estimated)	Time (spent)	Time (left)	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Project Managment / Documentation																	
Organizing	30	**33.5**	-3.5	4	2.5	1.5	3	3.5	1.5	2	2.5	2	2	2	3	2	2
Customer meeting	150	**116.5**	33.5	14	14	14	14	14	10.5	6	6	6	6	6	0	6	0
Report	350	**334**	16	24	18	13	2	8.25	9	24.5	65.75	3	0	17	14	43	92.5
External Communication	2	**1.5**	0.5				0.5	0	0	0.5	0.5	0	0	0	0	0	0
Scrum/Meeting	200	**216**	-16	21	21	27	21	15	17.5	14	7	7	20	6	15.5	8	16
Supervisor meeting	30	**35.5**	-5.5	11.5		7		7		6		4					
Gantt/Milstone plan	4	**2.5**	1.5			2.5											
WBS	3	**11**	-8		11												
Product Backlog	15	**21**	-6		21												
Declare expectations and goals	7	**8**	-1	8													
Requirments	25	**15**	10		4	3	8										
Burndown Chart	1	**1.5**	-0.5			1.5											
Responsibilities	3	**3.5**	-0.5	3.5													
Peer Evaluation	10	**12**	-2										12				
Analysis																	
Collaborative filtering research	5	**6**	-1			6											
Differences iOS/Android	3	**3**	0			3											
API test	4	**5**	-1			5											
Reuse Stedr	12	**2**	10				2										
Design																	
Design of architecture - report	8	**8**	0			8											
Database Sketch	2	**1.5**	0.5			1.5											
Mockup design	12	**18**	-6			8	10										
Papirprototype - utkast	5	**2**	3			2											
Architecture	3	**15**	-12	15													
Risk analysis	4	**2.5**	1.5	2.5													
Scenarios/Usecase	4	**12**	-8	12													

Task	Time (estimated)	Time (spent)	Time (left)	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Define the case	7	8	-1	8													
Startup with the framework	20	11	9			11											
Name	10	0.5	9.5			0.5											
Development																	
AngularJS	15	3	12						3								
Docker	10	38	-28				16			5	3	6	8				
Frontend/Backend	15	5.5	9.5							5.5							
Categorie Mapping	10	9	1				8	1									
FrontEnd																	
Misc	40	39.5	0.5						5.5	3	2.5		10.5	4.5	13.5		
Prototype	4	15.5	-11.5					7	4.5	4							
Onboarding	8	10.5	-2.5											7	3.5		
Login view	15	21	-6					8	2			4	7				
Story view	20	25.5	-5.5					11	5			3	3	3.5			
List view	15	14	1					8	3				3				
Recomendation	15	5	10								5						
Settingsview	15	12	3						3			2	7				
Preference View	15	9	6							9							
Polish	50	82	-32							10.5		12	12	15	8	11.5	13
Styling - Look	15	7	8						1		6						
Bug fixing	50	69.5	-19.5								5.5		5.5	3.5	9.5	18.5	27
Android	15	5	10										5				
IOS	15	3	12										3				
Backend																	
Misc	50	38	12							4.5			14	4.5	6	9	
Harvesting	20	28	-8					21	5			2					
Login backend (Authen)	10	0	10														
Models	20	4.5	15.5					1			3.5						
Collaborative Filtering	60	31.5	28.5						3.5				6	12	10		

Task	Time (estimated)	Time (spent)	Time (left)	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Contentbased Filtering	60	46	14							15	3	16	6	4.5	1.5		
Mahout	30	29	1							3	9	3	14				
Php to Angular	25	16	9						16								
States	10	2	8								2						
Lists	10	6	4							5	1						
Rating	10	0	10														
Bug fixing	45	81.5	-36.5						16			16	13	7	25	0.5	4
Database	8	7	1					7									
Database	4	9.5	-5.5				1.5		7	1							
Misc	7	15	-8					7					6	2			
AngularJS (DB)	5	26	-21					1	25								
PHP file (DB)	5	4	1					4									
Testing																	
Testing Plan	15	25.5	-10.5					1	11.5	7	6						
System Testing	15	8.5	6.5										0.5				8
Frontend Testing																	
Unit Testing	35	35	0								2	9	3	1	14	6	
Usertesting	30	30	0				14.5	2	13.5								
Evaluation of prototypig	15	14	1				14										
Backend Testing																	
Unit Testing	40	57	-17								6.5	7		10	11.5	22	
Integration test	20	18	2									7			11		
TOTAL	1820	1812	8	94	121	114.5	114.5	126.75	163	125.5	136.75	109	166.5	105.5	146	126.5	162.5
Weekly burnout	0	1812		130	130	130	130	130	130	130	130	130	130	130	130	130	130
Total time left (from estimate)			1820	1690	1560	1430	1300	1170	1040	910	780	650	520	390	260	130	0
Total time left (from spent)			1812	1718	1597	1482.5	1368	1241.25	1078.25	952.75	816	707	540.5	435	289	162.5	0
Instructions, feedback, requests and support at																	
http://xaviesteve.com/go/agilespreadsheet																	

APPENDIX E

TESTING

E.1 Unit test cases

Table E.1: Unit test cases presented by a testId, description of how the test should be performed, what input data to use and expected results.

Test case ID	Description	Input data	Expected results	Result
Unit 1: Database Helper				
UN1.1	Update one rating value in the database	Random userId, random storyId, updateValue: 2	The function should return true when running the database request.	Pass
UN1.2	Update rating values in the database	Random userId, random storyId, update Value : 5	The function should return true when running the database request	Pass
UN1.3	Update new tag[1] that user have created	Random userId, tagName: 'NyTestTag2'	The function should return true when running the database request	Pass
UN1.4	Update a user's tag which is connected to a story	Random userId, Random storyId, tagName: 'Les senere'	The function should return true when running the database request	Pass

[1] A tag is equivalent to a bookmark. Because of references to variable names such as tagName, this has not been changed in this section.

UN1.5	Update a user's action of rejecting a story	Random userId, Random storyId	The function should return true when running the database request	Pass
UN1.6	Update a story as recommended	Random userId, Random storyId	The function should return true when running the database request	Pass
UN1.7	Get rating on a story done by a user	Random userId, Random storyId	Should return row with the rating presented as a integer	Pass
UN1.8	Get the predefined tags connected to a user	Random userId	Should return row with the tags "Les senere" and "Les" which all users in the system is connected to. If this user has self made tags, they should be included	Pass
UN1.9	Get all the tags connected to a user	Specified userId: 105	Should return row with the tags "Les senere" and "Les" which all users in the system is connected to, included the added tag 'NyTestTag'	Pass
UN1.10	Get the tags connected to a story and a user	Specified userId: 105, specified storyId: 'DF.1295'	Should return row with the tag 'NyTestTag'	Pass
UN1.11	Get stories with timestamp	-	Should return rows with 167 storyId and a timestamp which indicate when the story was last changed	Pass

UN1.12	Get subcategories to a specific story	Random storyId	Should return a row with an array of subcategory Ids	Pass
UN1.13	Get story information	Specified userId: 105, specified storyId: 'DF.5220'	Should return row with an array with the keys: userId, storyId, explanation, rating, false_recommend, recommended_ranking, in_front-end_array, estimated_Rating	Pass
Unit 2: Database Story				
UN2.1	Fetch story	Specified storyId: 'DF.1812', specified userId: 105	Should return row with an array of the categories to the story and tags and tagName that are connected to the userId	Pass
UN2.2	Get recommended stories for a user	random userId	Should return 10 rows with stories, and they should include: userId, storyId, recommended_ranking, explanation, false_recommend, title, introduction, author, categories and mediaId	Pass

UN2.3	Get list of stories which is tagged by a user	Specified userId: 103, tagName: 'NyTestTag'	Should return rows with stories, and they should include: storyId, title, author, introduction, date, tagName, categories and mediaId	Pass
UN2.4	Get subcategories per story	-	Should return 167 stories with numericalId and subcategory ids	Pass
UN2.5	Get all stories with storyId, numericalId and categories	-	Should return 167 rows with storyId, numericalId and categories	Pass
UN2.6	Get states per story	Specified userId: 258, specified storyId: 'DF.1600'	Should return a row with stateId, numTimesRecorded and latestStateTime	Pass
Unit 3: Database User				
UN3.1	Add user	example mail: example@example.com	Should return a userId that is not null	Pass
UN3.2	Get user from Id	UserId: generated new Id.	Should return a userId from the database which is equal the userId input	Pass
UN3.3	Update user e-mail	UserId: generated new id, e-mail: newmail@example.com	Should return a e-mail from the database which is equal the e-mail input	Pass

UN3.4	Update user age	UserId: generated new Id , age group: 0	Should return a user model from the database, which includes the age group that is equal to the age group input.	Pass
UN3.5	Update user categories	UserId: generated new id, category preference: [2]	Should return a row with user model from the database with the category preference that is equal to the category preference input.	Pass
UN3.6	Get user from e-mail	UserId: generated new id, email:54@example.com	Should return a user model from the database, which includes the e-mail that is equal to the e-mail input.	Pass
UN3.7	Get user categories	UserId: generated new id, category preferences: [1,3,5,7,9]	Should return the categories from the database that is equal to the category preferences input	Pass
UN3.8	Get user e-mail from Id	UserId: generated new id	Should return a user model from the database,which includes the e-mail that is equal the e-mail input.	Pass

Unit 4: User Model				
UN4.1	Initiate User	userId:1, mail: example@example.com	Should create a user model with the correct input. The getters should return the values that matches with the input values	Pass
UN4.2	Set all user details	userId:1, mail: example@example.com,gender: 0, age group: 1, use of loc. : 0 , category preference: [1,3,5,7,9]	Should add the correct user details to the user model. The getters should return the values that matches with the input values	Pass
UN4.2	Print all	userId:1, mail: example@example.com,gender: 0, age group: 1, use of loc. : 0 , category preference: [1,3,5,7,9]	The printAll function should return a string with all the correct attributes and values that matches the input values .	Pass
Unit 5: Compute Preference Value				
UN5.1	Compute preference value for all stories for a user	Random userId	Should return 167 rows from the database, and every row should include: storyId, userId, numericalId and preferenceValue	Pass
UN5.2	Compute preference value of a story for a user	Random userId , specific 'DF.1098'	Should return an array with userId, storyId, numericalId and the preference value	Pass

UN5.3	Compute preference value	Specific storyId	Should return the preference value which is the type double	Pass
Unit 6: Recommendation				
UN6.1	Run recommender	Random userId		Pass
UN6.2	New content-based recommendation	Input: setUp.xml - a data model with userIds and connected preference values.	The recommender should return the correct number of recommendations when it should create new ones	Pass
UN6.3	Add content-based recommendation	Input: setUp.xml- a data model with userIds and connected preference values.	The recommender should return the correct number of recommendations when its adding recommendations to existing ones	Pass
Unit 7: Database connection with Java				
UN7.1	Insert recommendation	userId: 1, storyId: Df.1098, DF.1501, explanation: 0, false_rec. :0, ranking: 3,4, estimatedValue: 0,0	Expected table in XML-representation(insert-expected.xml) is the same as the actual from the database	Pass

UN7.2	Insert and update	userId: 1, storyIds: DF.1709,DF.1849, explanation:“updated”, false_rec. : 1,0, typeoffrec. :1,1, ranking: 3,4 estimated-Value: 4.5, 2.5	Expected table in XML-representation(insertUpdate-expected.xml) is the same as the actual from the databas	Pass
UN7.3	Delete recommendations	UserId: 1	Expected table in XML representation(delete-expected.xml) is the same table returned from DB	Pass
UN7.4	Rated	UserIds: 3,5 numericalId: 1812,1901	The getRated function should fetch the correct stories. The input rating should be the same as the returned ratings	Pass
UN7.5	Test front-end array	Numerical Ids: 1849, 1901	Get stories from front-end array should match the front-end array in XML format(setUp.xml)	Pass
UN7.6	Create explanation	Numerical Ids: 1098,1115,1501	The create explanation function should return a string with the correct storyIds and their titles.	Pass

Unit 8: UI-Log in				
UN8.1	Log in with an already existing user - Check if response is correct	existinguser@ mail.no	The user should get access and be directed to the recommendation view(main view)	Pass
UN8.2	Log in with a non-existing user - Check if response is correct	"newemail@ gmail.com"	The user should get access and be directed to the setup view.	Pass
UN8.3	Login a user with the wrong e-mail format - Check if response message is correct	"newemail"	The user should not get access and get a textual response from the system that the format of the input is wrong.	Pass
Unit 9: UI-Story View				
UN9.1	If story contains sound clip or video, check if these are presented properly	- story with video -story with sound clip	Should show the presence sound clip and video in the tabs. Should be able to handle the different video types, and be able to play them.	Pass
UN9.2	Give a rating - Check if the stars change color and response message are visible to the user.		Stars should change color when user performs rating, and the user should get a response message from the system which says that this story has been rated.	Pass

UN9.3	Give the story a new bookmark with a name -add this -Check if response message is visible to the user	"new bookmark"	When a new bookmark is made and the story is stored here, the user should get response message about this.	Pass
Unit 10: Settings				
UN10.1	Update profile with the wrong e-mail format	updateemail	The user should get a response message from the system that the input was not valid and that the user should try again.	Pass
UN10.2	Update profile with a e-mail that already exists in the system	"alreadyexistingemail @email.com"	The user should get a response message from the system that the input was not valid and that the user should try again.	Pass

E.2 Integration test cases

Table E.2: Integration test cases

Test case ID	Description	Input data	Expected results	Result
I.1	Simulate a HTTP request which will log in a user for the first time with an e-mail address. Call getUserFromEmail to the database	e-mail: 'testnr16@example.com, requestType: addUser	The HTTP request should return successful message included the newly created userId. The database function should return a row with the userId, mail, age_group, gender and use_of_location	Pass
I.2	Simulate a HTTP request which will log in a user for the first time without an e-mail address. Call getUserFromId to the database	e-mail:null, request type:addUser	The HTTP request should return a successful message included the newly created userId The database function should return the usermodel containing userId, mail, age_group, gender and user_of_location.	Pass
I.3	Simulate a HTTP request which will update a users profile. Call getUserFromId to the database. Check if the updates where correct	e-mail: 'testMail23@example.com', request type: updateUser,	The HTTP request should return a successfull message. The database function should return a usermodel where the e-mail match the input.	Pass

I.4	Simulate a HTTP request that will create a new test user without an e-mail. Simulate another HTTP request which will update e-mail to this test user, with an email that already exists in the system Call the getUserFromId function to the database. Check if the user is not updated	e-mail: 'testMail23@example.com'	The HTTP request should return a failure message. The database function should return an usermodel where the e-mail is null.	Pass
I.5	Simulate a HTTP request that will create a new test user without an e-mail. Simulate another HTTP request which will update e-mail to this test user Call the getUserFromId function to the database. Check if the user is updated		The HTTP request should return a successful message. The database function should return an usermodel where the e-mail matches the input.	Pass
I.6	Simulate a HTTP request that will create a new test user with an e-mail Simulate another HTTP request that will return the user model from the users e-mail Check if the returned data has all attributes required, and that the data match with the input data	e-mail: 'getUserFromEmailTest@example.com", request type: addUser/getUserFromEmail	The HTTP request should return a usermodel with the attributes userId, e-mail, age_group, gender, use_of_location and with the data which match the input data.	Pass

I.7	Simulate a HTTP request that will create a new test user without an e-mail. Simulate another HTTP request that will return the user model from the users id. Check if the returned data has all attributes required, and that the data match with the input data	e-mail: null, request type: addUser/getUserFromId	The HTTP request should return a usermodel with the attributes userId, e-mail, age_group, gender, use_of_location and with the data which match the input data.	Pass
I.8	Simulate a HTTP request that will create a new test user without an e-mail. Call the getNumberOfRatingsDoneByThisUser to the database. Simulate another HTTP request which will connect a rating to a story for this user. Call the getNumberOfRatingsDoneByThisUser to the database again Check if another rating is registered	e-mail: null, request type: rating, storyId: 'DF.52201, random userId	The HTTP request should return a usermodel with the attributes userId, e-mail, age_group, gender, use_of_location and with the data which match the input data.	Pass
I.9	Simulate a HTTP request that will create a new test user without an e-mail. Simulate another HTTP request where the user rejects a story. Check if the rejection is stored	e-mail: null, request type: rejectStory		Pass

I.10	Simulate a HTTP request that will create a new test user without an e-mail. Simulate a request that will create a new tag for this user Simulate a HTTP request that will return the list of tags for this user Check if the new tag has been added	request type: addUser/addNewTag/getList tagName: 'newTag1'	The returned list should only include the one tag that where created.	Pass
I.11	Simulate a HTTP request that will create a new test user without an e-mail. Simulate a HTTP request that will create a new tag for this user. Simulate a HTTP request that will return the list of stories contained in this taglist. Check if the new tag has been added	request type: addUser,tagStory,getList tagName: 'newTag1', storyId: 'DF.5223'	The returned list should have the recently added story in the top of the list. The list should have the following attributes connected to every story: id, title, description, false_recommend, explanation, picture, thumbnail, categories, mediaType, author, data	Pass

I.12	Simulate a HTTP request that will create a new tag for a user. Simulate a HTTP request that will return all the tags connected to this user. Check if the tag was stored Simulate a HTTP request that will remove the new tag that were created earlier Simulate another HTTP request that will return all the tags connected to this user Check if the new tag has been removed	Specified userId: 105, request type: addnewTag, getAllLists, removeTag, tagName: 'tagToBeRemoved', storyId: 'DF.6081'	The returned list should have the recently added story in the top of the list. The list should have the following attributes connected to every story: id, title, description, false_recommend, explanation, picture, thumbnail, categories, mediaType, author, data	Pass
I.13	Simulate a HTTP request that will return the list connected to a tag for an user	Specified userId: 105, request type: getList, tagName: 'Les senere'	Should return a list of the stories connected to this tag. The stories should have the following attributes: id, title, description, false_recommend, explanation, picture, thumbnail, categories, mediaType, author, date	Pass
I.14	Simulate a HTTP request that will return all the tags connected to a user	Specified userId: 105, request type: getList, tagName: 'Les senere'	Should return a list of the tags. The tags should have the attributes text and checked.	Pass

E.3 System test cases

Table E.3: System test case for creating a recoverable profile.

Test ID	T1
Test Item	Create recoverable profile
Approach	The user locate and press the “register user” button in the app. Applies the e-mail in the correct format. The response is valid and the user gets feedback.
Input data	“newuser@example.com”
Expected results	The user writes the correct e-mail address and get the correct feedback from the system: ”Kontakter server” and will be directed to the startup page.
Testing task	1. Click “create user”-button. 2. Apply e-mail address to the e-mail input field 3. Receive feedback from the system 4. Check e-mail inbox to se if the correct mail from the system was received
Depends on tests	
Pass/Fail	Passed

Table E.4: System test case for login with e-mail registration

Test ID	T2
Test Item	Log in with e-mail registration
Approach	The user locate the login button, applies the registered e-mail and obtain access to the system and the profile connected to this e-mail address .
Input data	valid email: “user@example.com”,example invalid e-mail: “mail@example”
Expected results	• The first time the user have logged in: System Response: Choose preferences-view should appear. • The user have done this process before: System Response: ”Vennligst vent mens vi finner historier vi tror du vil like” and direct the user to the view with the recommended stories. • The user types an e-mail with wrong e-mail format: System Response: ”Ikke en gyldig adresse”
Testing task	1. Navigate to the login view 2. Apply e-mail address to the e-mail input field 3. Receive response from system
Depends on tests	T1
Pass/Fail	Passed

Table E.5: System test case for initial settings

Test ID	T3
Test Item	Set initial settings
Approach	The user is logged in to the system for the first time. The user will choose age group and gender in the initial settings that will appear the first time the user is logged on to the system. After this the user will be asked to choose cultural category preferences.
Item pass/Fail criteria	
Input data	User action: click the buttons for the age group, gender and interests, and next buttons
Expected results	The user will click the buttons for age group, gender, and interests. The buttons will change color when clicked on. The user navigates to next view by clicking the next button and will obtain access to the system. If the user do not select interests the user will get a response for the system that it is necessary.
Testing task	1. Start app 2. Click the correct age group and gender. 3. Navigate to the next page 4. Navigate to the next page without selecting interests 5. Receive feedback from system 6. Select two interests 7. Navigate to the next page
Depends on tests	
Pass/Fail	Passed

Table E.6: System test case for browsing recommended stories

Test ID	T4
Test Item	Browse recommended stories
Approach	The user is shown a list of recommended stories. The user will click on the first story. The story will be viewed and the user closes it.
Item pass/Fail criteria	
Input data	User action: click arrow
Expected results	The view should show a list with recommended stories. The view of the stories will include a title, story picture or default picture, an introduction, category icons, media icons and a explanation of why this story is recommended to the user. The view should show 10 more recommendations when the user have browsed through the 10 first stories in the list. When the user clicks a story the full story should appear in a full screen, including text and potential pictures, videos and sound clips.
Testing task	1. Click on a recommended story 2. Locate back button and close the story 3. Click on the next recommended story 4. Navigate
Depends on tests	T1,T2
Pass/Fail	Passed

Table E.7: System test case for adding a story to list

Test ID	T5
Test Item	Add story to list
Approach	The user navigate to a story and gives the story a rating. The user clicks the bookmark button in the story, and gives a name to a new list of stories.
Input data	User action: Click on a star, Give name to a new list of stories: ex. "My Favorite Stories"
Expected results	The story that was rated with stars are automatically put in the "read" list. The story should be stored in the new list of "My Favorite Stories" and the user have access to this by navigating to the menu, and then to "Bokmerker".
Testing task	1. Click on a story in the recommendation view 2. Click on the star icon to in the right upper corner 3. Click on one of the stars in the rating view to give rate. 4. Click out from the view. 5. Click the bookmark button in the story 6. Click the plus icon 7. Type in a "My Favorite Stories" and click out of the view. 8. Navigate to the lists via the menu 9. Click on "My Favorite Stories" 10. Click on the first story in the list 11. Navigates to the "read" list 12. Click on the first story in the list
Depends on tests	
Pass/Fail	Passed

Table E.8: System test case for giving rating

Test ID	T6
Test Item	Give a rating
Approach	The user gives a story a rating by clicking the stars. The user navigates to bookmark lists and checks if the story was stored in the "read" list
Input data	User action: clicks on story, navigated to rating view, clicks on a star.
Expected results	The star buttons that were pressed will change color. The system should store the rating for that story. The next time the user clicks on this story - the yellow stars will show the users rating.
Testing task	1. Click on a story in the recommendation view 2. Click on the star icon to in the right upper corner 3. Click on one of the stars in the rating view to give rate. 4. Click out from the view. 5. Click on the star icon again 6. Close rating view
Depends on tests	T1,T2
Pass/Fail	Passed

Table E.9: System test case for specifying settings

Test ID	T7
Test Item	Specify settings
Approach	The user navigates to ”Innstillinger” via the sidebar menu. The user then adds preferences and change the permission to use location.
Input data	User action: click the buttons for the age group, gender and interests.
Expected results	The user will click the buttons for age group, gender, and interests. The buttons will change color when clicked on. The user navigates to next view by clicking the next button and the system will give the response: ”Vennligst vent mens vi finner historier vi tror du vil like”. The system saves the users preferences and updates the recommended stories list. The list of stories should now be updated.
Testing task	1. Navigate to settings 2. Navigate to preferences 3. Add another preferences 4. Check the recommended stories list to see if it is updated
Depends on tests	
Pass/Fail	Passed

E.4 Usability test cases

DNS: Desired next step with reference to the numbering in the View column

Table E.10: Usability test

View	DNS	Comments
1. Intro / Tutorial	2	**Feedback:** Not clear if user is supposed to swipe or not. The view is lacking in content. Users do not know if they can go back between views. The users would like an introduction to the application. **Changes:** Add animation/intro
2. Login	3	**Feedback:** Users often click the skip button instead of logging in. The description of why users have to input their e-mail is not clear. The text is too small. The term ”email verification” is not understood. **Changes:** Change wording in the view. Make the skip button smaller to encourage logging in instead.
3. Profil	4	**Feedback:** Some users thought this was part of the recommendation. They wonder if this is for research data or for the application. **Changes:** Explain why age and gender information is needed. Make the default age and gender not selected so user is not forced to choose.
4. Interests	5	**Feedback:** Users do not understand what this view is, they want an explanation. It is not clear if you can choose more than one interest. **Changes:** Add a better explanation.
5. Media format	5	**Feedback:** The sound icon was interpreted as music. User did not understand what they were choosing, and how this would affect stories. **Changes:** Remove this view, because it does not add much to the application, and users are confused about its function and meaning.

6. Recommendations	6	**Feedback:** The read later button is not understood. Users do not understand that it places the story in a list. Users do not understand that they can swipe in this view, or what they can touch. The users feel a bit ”dumped” into this view with no explanation. **Changes:** Remove read later button and add a bookmark icon at the top right instead. Add a card deck or something else to show users that they can swipe. Add a loading animation for when the view is retrieving stories. Change the title of the view to ”Anbefalte historier”. Make it clear that the card can be touched and make it possible to remove the card.
7. Story View	7	**Feedback:** Some users did not understand the bookmark icon. They did not know where to go from this view. There was also some confusion about what the ”ikke interessert” button means. **Changes:** Add a back button. Change the name of the ”ikke interessert” button. Make the icons more understandable.
Story bookmark	8	**Feedback:** Users want to know if they can change the name of existing lists. **Changes:** None
8. Menu	9	**Feedback:** User does not understand what the ”utforsk” button mean. **Changes:** Change the name of ”utforsk” to make it clear that this is where the main personalization view is.
9. List	9 >12	**Feedback:** Some users are unsure how to delete a story from a list. They try to swipe in both directions. **Changes:** Add some hint or indicator that you can swipe. Maybe let users swipe both ways.

APPENDIX F

MEETING REPORT EXAMPLES

Meeting with the Group

Group id: 8
Date and time: 03/03 -15
Location: B1-126

Required Information

From the group members:

- Status

Agenda

1. Status

- Audun: Skrevet kode for å legge inn historie i db. Æøå-problem, ellers går det bra.
- Espen: Jobbet videre med list view, mesteparten er ferdig. Har splittet opp rapporten i kapitler. Har nå ansvaret for å copy-paste inn i en samlet en på slutten av hver uke. Kommentarer og endringer skal nå skje i smådokumentene.
- Ragnhild: Fikset på story view, lagt til rating-mekanisme. Sliter med å få til video og audio på Android.
- Hanne Marie: Skrevet litt på content-based og collaborative filtering. Trenger mer oppgaver.
- Kjersti: Sende over data fra PHP til Angular. Sitter litt fast med det. Har snakket med ny supervisor, hun ville ha rapport og status før fredagen. Har jobbet litt med testplan.
- Roar: Har drevet med evaluering av brukertester. Testet på to personer til i helga. Skrevet litt om hva som er gjort og hva som ble funnet under testingen. Skal skrive noe om brukertesting i rapporten. Har også hatt problemer med Angular. Organisering med burndown - skal det legges ut på Facebook etter hver sprint slik at folk ser det? Har også begynt å skrive om hver sprint i rapporten.
- Eivind: Har tenkt på hva som skal gjøres med kommunikasjon frontend og backend. Vi må velge hva som skal sendes til frontend (id og tags og slikt eller alt sammen?)
  - Rating, id, tags, kategorier, lister er minimumskrav.
  - Hva skal hentes når den startes opp / se på lister / story view?

Mail: Shan shan vil gjerne se repository på frontend.

Additional Information

To do

Meeting with the Customer

Group id: 8
Date and time: 9/2 -15
Location: SINTEF

Required Information

From the group:

- Action Points:
    - Framework
    - Requirements
    - Architecture
    - Architecture - Diagram

From the customer/supervisor:

Agenda

1. Godkjenning av forrige møtes referat
2. Avtale møte med SINTEF for å lære om deployment of database og server side system
    a. Starte med å lage lokal server med docker. Vi får en bruker til SINTEF-server.
3. Prioritere funksjonelle krav
    a. Spesifisere kravene mer
4. Krav/egenskaper
    a. Alder/kjønn er det viktig i forbindelse med forskning, ikke i forbindelse med personalisering.
    b. Veldig viktig med kulturelle preferanser
    c. Lokasjon low/medium prioritet
    d. Media preferanser er viktige
    e. Appen skal minne folk på at de må gi tilbakemelding på historier.
    f. Gi anbefalinger som brukeren kanskje ikke liker. Si ifra til brukeren at han kanskje ikke liker denne historien, men at det kan være en interesant historie. Unngå å sette bruker i bås.
    g. Mulighet for å velge enten swipe eller liste. Swipe mulig i flere moduser (?): swipe gjennom recommended stories, swipe gjennom lagrede historier osv.
    h. Bruker kan starte app selv, eller få notifikasjon til et bestemt tidspunkt med historie
    i. Man har tre valg på browse stories: reject, save for later og read.
    j. Definere historikk
        i. Bruker kan sette egne tags på ting han har lest. For lettere å kunne finne igjen historier. (Brukertags) [sortere på stjerner (hvis det brukes)]
        ii. Mulighet for å lagre favoritt/bokmerke?
            1. Må spørre testbrukere om de vil ha bokmerke.
        iii. Lagre liste med historier bruker ønsker å lese senere
5. Ikke-funksjonelle krav
    a. Et krav som er veldig viktig: usability. Appen skal være lett å bruke.
        i. Utføre usability test
    b. Hvis vi skal ha username og passord er sikkerhet viktig. Hvis vi sikkerhet ikke har høy prioritet, bør vi kanskje droppe username og passord.
    c. Restrict testing til bare områder i Trondheim? For at det skal bli lettere å bruke collaborative filtering. Begrense antallet historier.

APPENDIX G

DEVELOPER GUIDE

G.1 Front-end

G.1.1 Dependencies

Android

- Android SDK (See the Requirements and Support section at https://cordova.apache.org/docs/en/3.3.0/guide_platforms_android_index.md.html#Android%20Platform%20Guide)

To sign apk file:

- Keytool (JDK)
- jarsigner (JDK)
- zipalign (JDK)

iOS

Only possible on Mac OSX

- Xcode (install from the App Store) Minimum required version is Xcode 4.5. More on this at https://cordova.apache.org/docs/en/3.3.0/guide_platforms_ios_index.md.html#iOS%20Platform%20Guide

G.1.2 Setup Guide

1. Install NodeJS (https://nodejs.org/download/)
2. Install Cordova and Ionic (http://ionicframework.com/getting-started/)

   ```
   npm install -g cordova ionic
   ```

3. Install Gulp (https://github.com/gulpjs/gulp/blob/master/docs/getting-started.md)

```
npm install -g gulp
```

4. Restore State (Install dependencies)

   ```
   ionic state restore
   ```

G.1.3 Testing

Use these commands in the root folder of the project to run the application.

Android

First run:

```
ionic build android
```

- Emulator (Note that the default Android emulator has many problems and is not recommended to use)

  ```
  ionic emulate android
  ```

- Device

  ```
  ionic run android
  ```

Read more about ionic testing here: `http://ionicframework.com/docs/guide/testing.html`

iOS

First run:

```
ionic build ios
```

- Emulator (You need to run this to install the emulator: `npm install -g ios-sim`)

  ```
  ionic emulate ios
  ```

- Device (requires Apple Developer account)

  ```
  ionic run ios
  ```

- Ionic View app

  You can install an app on iOS called Ionic view, and test ”Vettu Hva?” through it. Requires an Ionic user.

  Run:

  ```
  ionic upload
  ```

  You will then be asked to enter login information.

  Then it should be possible to download and run in Ionic View. If it for some reason should not appear, click on the eye icon at the top left and enter the ID from the command line.

  More information about Ionic View: `http://blog.ionic.io/view-app-is-alive/`

Browser

The application requires internet access. Testing this way is not recommended. The app uses cordova plugins, and these will not work in the browser.

- Both platforms (Android and iOS) in one view

  ```
  ionic serve --lab
  ```

- Single view which can be resized

  ```
  ionic serve
  ```

G.1.4 Building

Android

1. Generate release build

   ```
   cordova build --release android
   ```

2. Generate key

   ```
   keytool -genkey -v -keystore my-release-key.keystore -alias alias_name
   -keyalg RSA -keysize 2048 -validity 10000
   ```

3. Signing the APK

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1
-keystore my-release-key.keystore HelloWorld-release-unsigned.apk alias_name
```

   Where the AKP is located i.e. `StoryTelling-Frontend\platforms\android\ant-build`

4. Optimize the APK

```
zipalign -v 4 HelloWorld-release-unsigned.apk HelloWorld.apk
```

(These steps are necessary to update the application after you published it the first time)
PS. Save the keystore file generated in step 2 for further patching.
Read more about publishing Ionic applications on Android here: `http://ionicframework.com/docs/guide/publishing.html`

iOS

Apple Developer account required. You can open the .xcodeproj project from the platforms/ios folder.

You need to set up Xcode with your certificates: `https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertific`
`html#//apple_ref/doc/uid/TP40012582-CH31-SW6`

How to distribute the app to test users: `https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/TestingYouriOSApp/TestingYouriOSApp.html#//apple_ref/doc/uid/TP40012582-CH8-SW1`

G.2 Back-end

G.2.1 Dependencies

PHP

Choose between installing and configuring PHP or downloading a development environment with PHP and MySQL.

- Install and configure PHP: http://php.net/manual/en/install.php
- Development environment alternatives:
  - XAMPP: https://www.apachefriends.org/download.html
  - WAMP for windows: http://www.wampserver.com/en/#download-wrapper

MySQL

- MySQL workbench: https://dev.mysql.com/downloads/workbench/

Java

- Download Java Development Kit: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Docker

- Docker install: https://docs.docker.com/installation/

G.2.2 Setup Guide

1. Install PHP, MySQL and Java on preferred development device
2. Install Docker on chosen server
3. Clone "Vettu hva?" repository:

   ```
   git clone https://github.com/ewolden/vettu-hva
   ```

4. Copy config.php and Dockerfile to server.
5. Define the paramaters in the config.php file:
   - DB_USERNAME - Can be left as root since this is only used internally in the image
   - DB_PASSWORD - Can be left blank since this is only used internally in the image
   - DB_HOST - Can be left blank which is the same as typing localhost
   - DB_NAME - storytelling is the name of the database
   - API_URL - This is the URL to digitalt musems API
   - API_KEY - The API key needed to use digitalt museums API

- APP_MAIL - The email used for sending users emails upon creation of user (uses a gmail address)
- APP_MAIL_PASS - The password for the email given

6. Set up docker container on server

    - Run the following command to set up a docker container (see more detailed description here: `https://github.com/ewolden/vettu-hva`):

      ```
      docker build -t optional_ImageName folder_with_config_and_dockerfile
      ```

    - Create a volume for docker to store databases in (only needs to be done the first time):

      ```
      docker create -v /dbdata --name Storytelling-DBdata optional_imageName
      ```

    - Start the docker image and from the database storage:

      ```
      docker run -d --volumes-from vettuhva-DBdata --name vettuhva-backend -p
      optional_external_port:80 -p optional_external_port:3306 -e
      MYSQL_PASS="chosen password" optional_imageName
      ```

    - To stop the running container:

      ```
      docker stop containerName
      ```

    - To remove old containers:

      ```
      docker rm containerName
      ```

G.2.3 Testing

Use PHPUnit for testing PHP code. Download and get started with PHPUnit here: `https://phpunit.de/getting-started.html`
Use JUnit for testing Java code. Get started with JUnit here: `https://github.com/junit-team/junit/wiki/Getting-started`

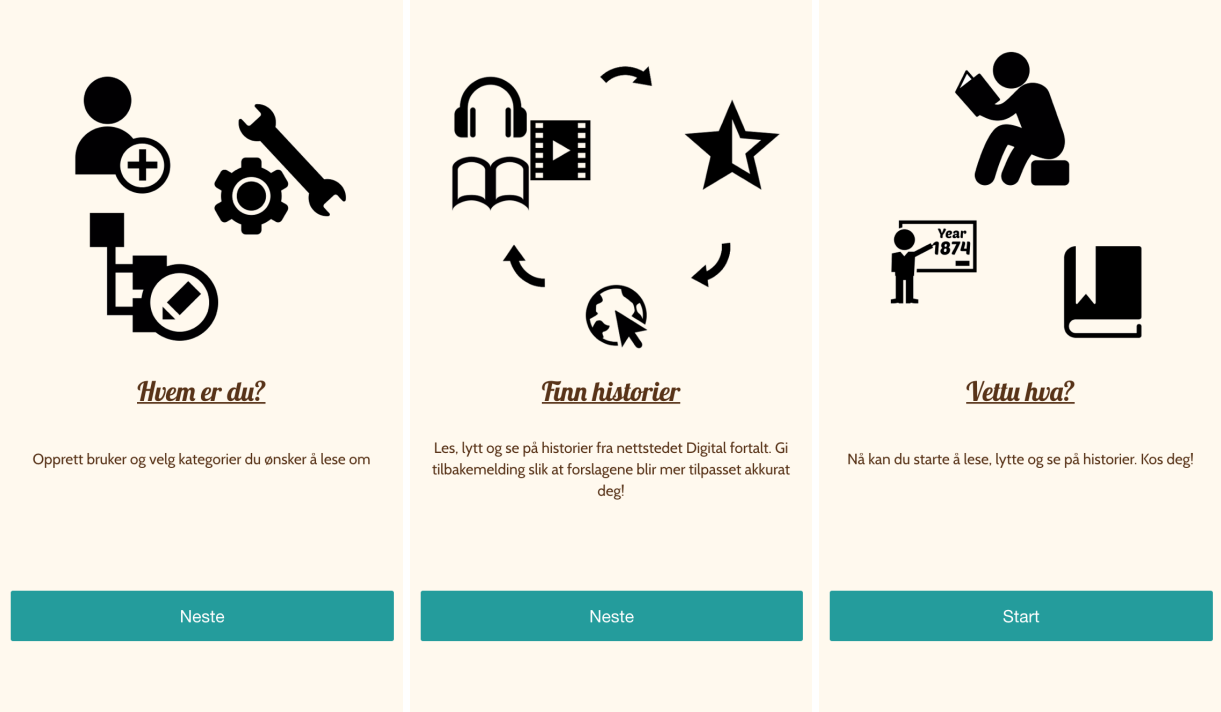
APPENDIX H

USER MANUAL

This is a guide on how to use the application ”Vettu Hva?”. It will explain the the basics of how to interact with the app.

H.1 Starting the app

When opening the app for the first time, an introduction to the app will be displayed. This can be browsed by tapping the ”Neste” button.



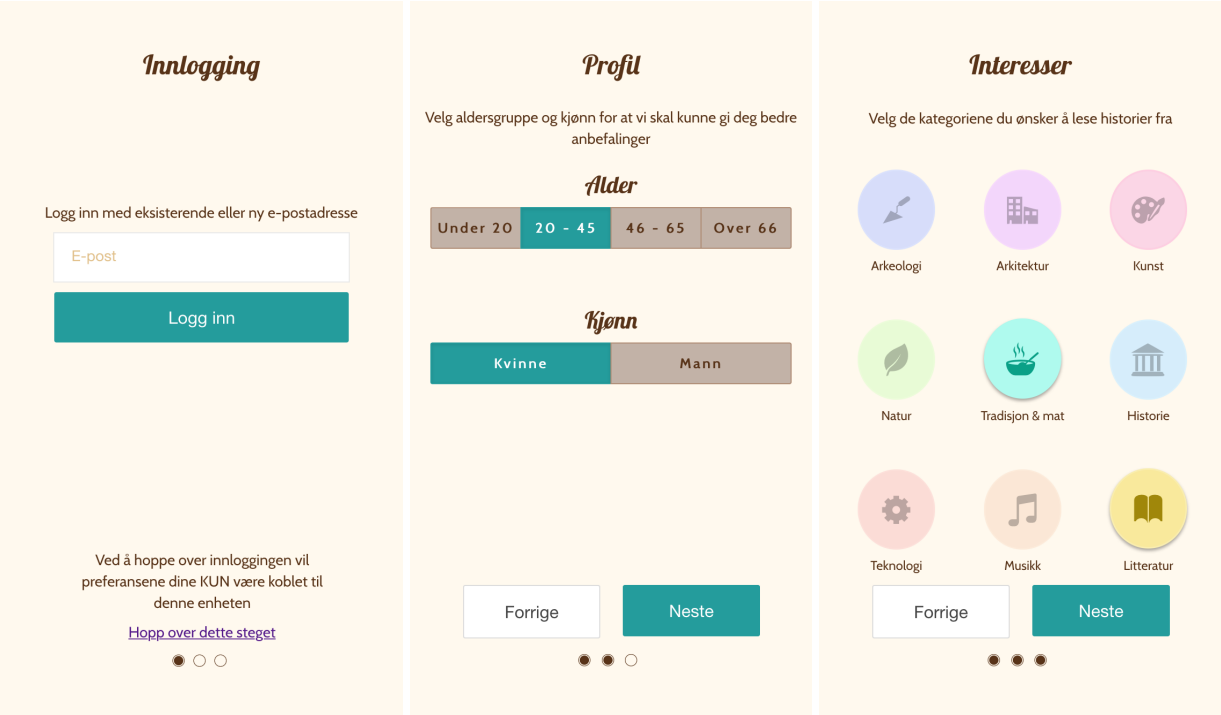
(a) (b) (c)

Figure H.1: Introduction

After the last of the three introduction screens, the login screen is displayed. There are three choices:

- Create a new user by entering an e-mail address that have not been registered in this application before. A confirmation e-mail will be sent to this address.
- Log in to an existing user by entering the e-mail address connected to that user.
- Skip this step by tapping the link towards the bottom of the screen. This means that there is no possibility to log in on another device, and if logging out there is no way to retrieve the information associated with the user. This can be done at a later time in settings by entering an e-mail address.

If logging in, the main screen will be displayed. Otherwise, the app will additionally ask for profile information (age group and gender) and cultural interests.



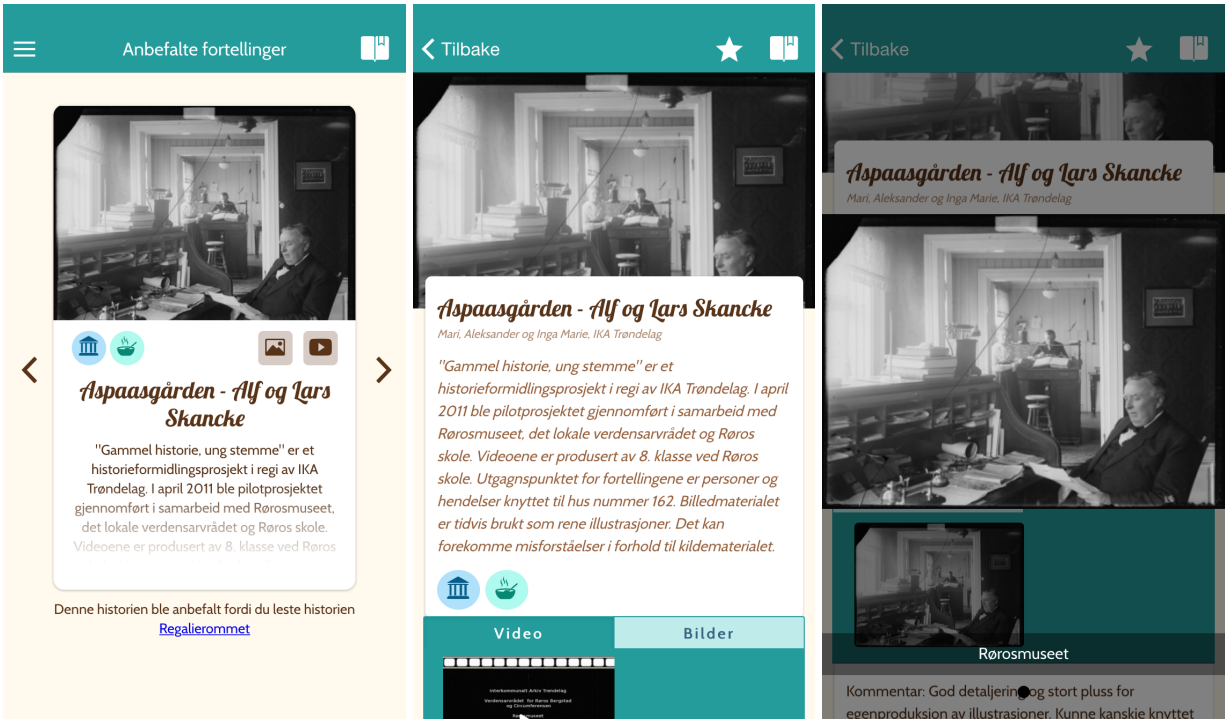
(a) Sign in **(b)** Profile **(c)** Interests

Figure H.2: Login

H.2 Browsing stories

Browse stories by swiping left and right or by tapping the arrow buttons on each side. Under the photo of each story are the categories the story belongs to and the media types that the story offers. The text below each card is the explanation of why the story was recommended. This may not appear in the beginning as it will not have any feedback to be based on. Tap a card to read the story.

The detailed view of a story displays text content, images, video, and/or audio. Which media types are available will vary from story to story. Images and video can be viewed in fullscreen, except some videos which are from Youtube or Vimeo which will be opened in the browser.

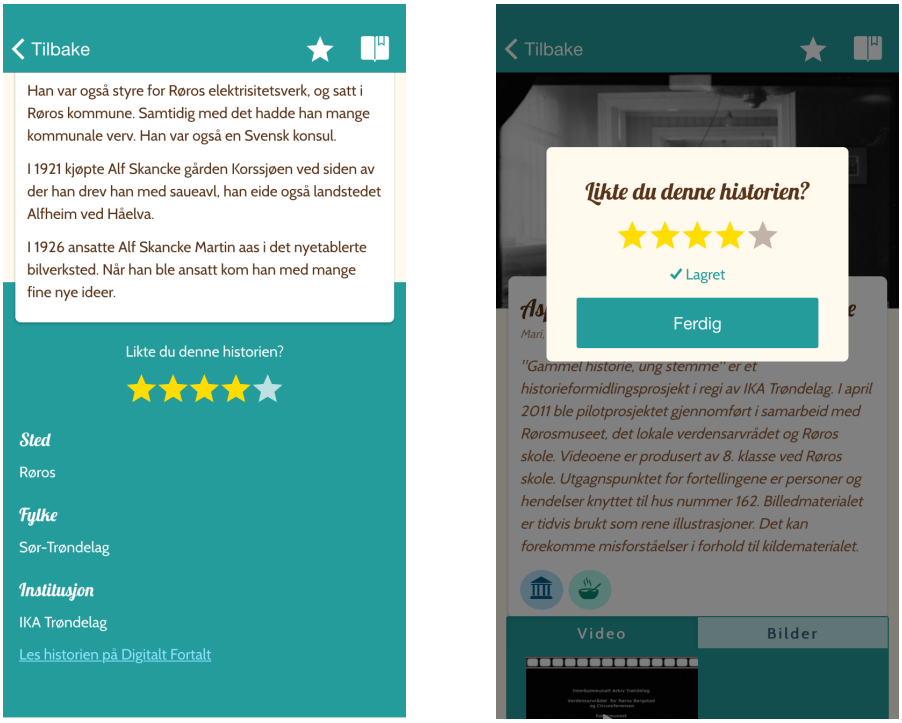


(a) Recommended stories **(b)** Detailed view of a story **(c)** Fullscreen image

Figure H.3: Browsing stories

H.3 Rating a story

After reading a story, give the app feedback by rating the story. This can be done either by tapping the stars at the bottom of the story, or by doing it in the popup that appears when tapping the star icon in the top bar. This feedback is necessary for the app to provide good story recommendations.

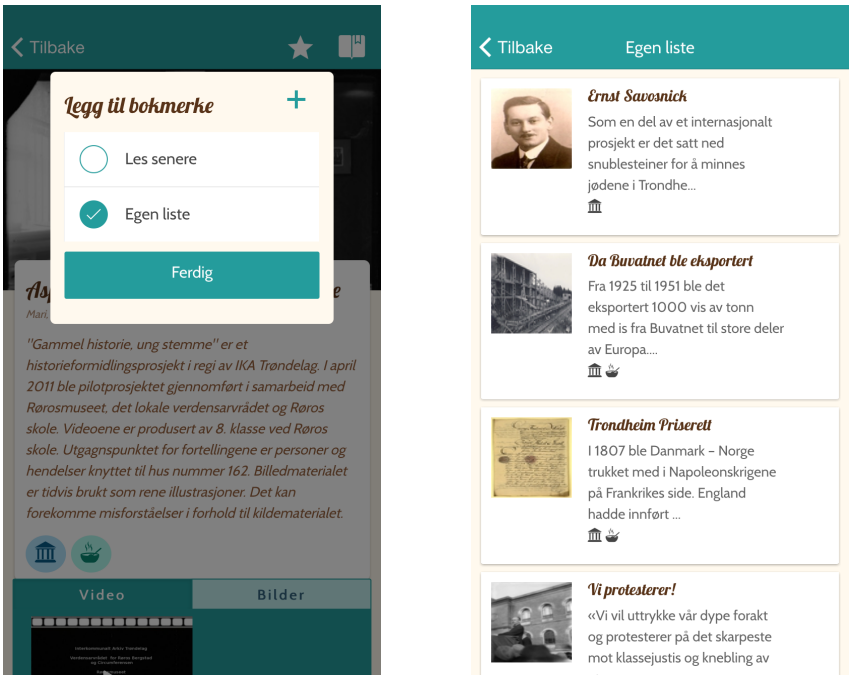


(a) Rating at bottom of a story **(b)** Rating popup

Figure H.4: Rating a story

H.4 Bookmarks

To save a story for later bookmarks can be added, both when browsing recommendations and when viewing the details of a story. This is done by tapping the bookmark icon in the top bar. A popup will appear with the predefined ”Les senere” list. To create a new bookmark list, you can tap the plus icon and type in a name. The story can then be found later by going into the menu and selecting the list it was saved in. When viewing a list, a story can be removed from it by swiping to the left. To remove an entire list, go to the menu and tap the x-icon on the list to delete.

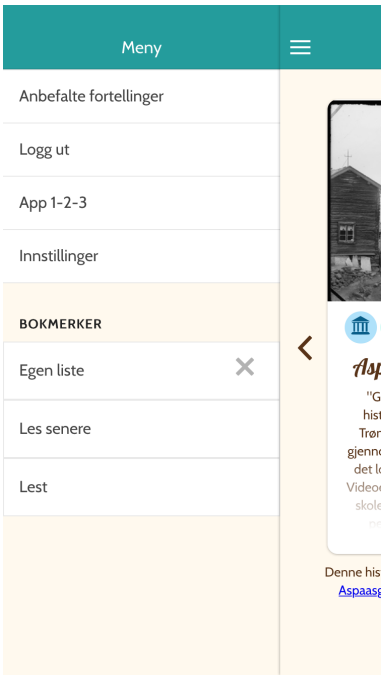


(a) Add bookmark **(b)** Bookmark list

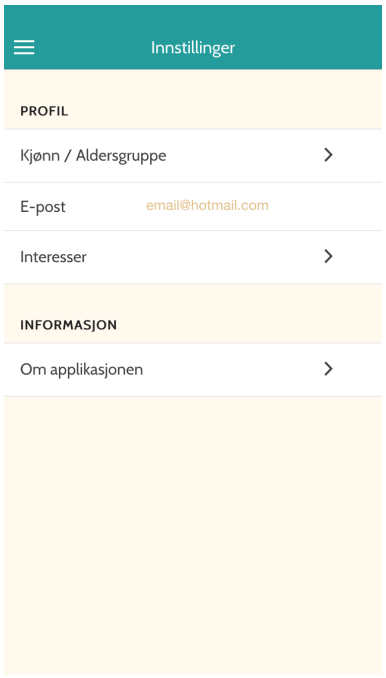
Figure H.5: Bookmarks

H.5 Other

From the menu it is possible to view bookmarks, settings, go through the app introduction again or log out. In the settings the email address, profile information and interests can be changed. It also provides more information about the app and the project.



(c) Menu



(d) Settings