# A Helping Hand for the Physical Stores

Anders Emil Nielsen & David Harboe


Kongens Lyngby 2013
B.Sc.-2013

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
compute@compute.dtu.dk
http://www.compute.dtu.dk  B.Sc.-2013

# Summary

The thesis deals with the development of a scalable and robust system with a mobile platform that can be used as a marketing platform for the physical shops. This mobile platform is an attempt to support the physical shops by attracting customers into their shops. The idea was developed into an iPhone app with a focus on user experience. The platform need to be able to handle many users, which means that the system behind the platform needs to be scalable and robust in order to have meet the user experience requirements. To develop such a system, the technologies used for the system have been well-considered with focus on both the requirements and time to market for the product. The iPhone app was developed as a native app with Apple's environment. The web service was implemented with Ruby on Rails and the NoSQL database MongoDB, because of its ability to scale horizontally. In order to verify the requirements of the system, user and load testing was performed. The results of the user tests showed that the platform gave a unique experience with a few glitches. The load tests showed that scaling with the Platform as a Service provider, Heroku, was easier than with Cloud.dk, which was originally chosen. The product of this thesis did not meet all the non-functional requirements, but the system is prepared for scalability and will after some adjustments be ready for the App Store.

A short video presentation of the iPhone app is shown at
`https://www.dropbox.com/s/ons2botut31bnf1/mymodo_app_presentation.mov`

# Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring a B.Sc. in Informatics.

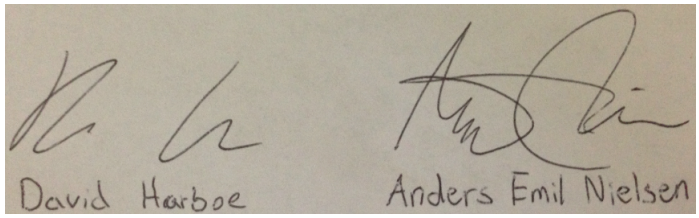
The thesis deals with developing a scalable and robust system with a mobile platform that has the goal of helping the physical stores with getting customers.

The thesis consists of a report and an implementation of a web service and iPhone app. The source code is on GitHub, but on private repositories. Access to these repositories can be acquired by emailing either davidharboe@gmail.com or aemilnielsen@gmail.com. The source code is also available for direct download. The download links are:

The web service:
GitHub: https://github.com/anderslime/mymodoservice
Direct download: https://www.dropbox.com/s/pdwwuoiyu5b2ult/mymodoservice-master.zip

The iPhone app:
GitHub: https://github.com/anderslime/mymodoapp
Direct download: https://www.dropbox.com/s/dc8jk1q4dxhsnv0/mymodoapp-master.zip

Lyngby, 1-July-2013




Anders Emil Nielsen & David Harboe

# Acknowledgements

We would like to thank our supervisor Joseph Kiniry for given clearance over the whole project. Thanks to Peter Harboe for supporting with the structure and guidelines of the report. Thanks to user test participants Mikkel Bjerregård, Mads Øhlenschlager, Nikolai Eskild Jensen, Lars Fischer Sjælland and Andreas Graulund.

# Contents

Chapter 1

# Introduction

The physical shops in Denmark have been losing revenue in the last five years. Even though the consumers have more resources, the shops have lost almost 15% of the revenue in the last five year. [Ras13] Looking at the US retail sector as a leading indication of the development, online shopping has increased turnover from 2002 to 2008 by 215.8% where retail shopping in physical shops only increased from 2002 to 2008 by 23.6%. [Lie11, table 1, page 38]

There is an opportunity in supporting the physical shops because value in buying offline still exists. The customer can feel and see the item in the real life. It is easier to return the item if something is wrong. The customer gets the item as soon as he or she leaves the shop, does not have to wait for postage, and at the end the customer gets a live experience. By combining on and offline shopping this value is kept.

## 1.1 Problem Description

This project is an attempt to make a mobile platform for the physical shops to reach out to numerous of customers with a specific deal and make it attractive to buy offline again with focus on the live experience.

The first mobile platform is an iPhone app with an interface for the physical shop and their customers. The physical shop is able to display a deal by taking a photo of their product, which is then shown to nearby customers on their iPhone. These deals have a time limit, which gives the user a sense of urgency and a live experience. Additionally the users are able to rate and comment on the deal and the physical shops. This gives value to other users, who then know the quality of the deals and shops.

## 1.2 Bachelor Formalities

This is a 40 (2 $x$ 20) ECTS point bachelor thesis, which results in a estimated time of 1120 hours. The work of the thesis has been divided between the two authors of the project. The iPhone app, web service and report must be delivered the 1st of July 2013. Before the start of the project, the criteria of success was settled as follows:

- Learn about the process of developing an iPhone App (from idea to shipping)
- Learn about distributed systems with regards to developing scalable web services for the cloud (architecture, database and caching)
- Develop an iPhone app with focus on usability and robustness, which fulfills the requirements in the project description
- Develop a web service, which should be able to persist and serve data for the iPhone app with resources needed for the requirements in the project description.
- The web service should be prepared for scalability
- The web service should require authorization

Chapter 2

# Requirements

The project started as only an idea and was, in collaboration with a team of five people including us, processed into concrete features for the iPhone app. This process is explained in more details in chapter 6. This chapter will analyse the problem stated in section 1.1 and set the requirements of the project. The requirements are divided into functional and non-functional requirements.

## 2.1 Functional

The features available in the app are dependent on which role(s) the user has. When using the app for the first time, the user will only have the role as an end user. An *end user* is then able to register and login as a *member*. As a member and a owner of a physical shop one is able to register their shop and become a *business user*. The functional requirements of the app can thereby be divided into, which role the end user has.

The functional requirements for an end user is as follows:

An end user must be able to...

- register as a member
- login as a member
- see the nearest deals
- see deals near an arbitrary location
- get a random deal near the location of the user
- see the latest comments and ratings near the location of the user
- see the nearest businesses based on the location of the user
- see the nearest businesses based on search parameters and an arbitrary location

A member must be able to...

- rate a given deal
- comment on a given deal with a text and possibility to attach a photo taken from the iPhone
- rate a given business
- comment on a given business with a text and possibility to attach a photo taken from the iPhone
- register a shop and become a business user

A business user must be able to...

- see own created deals
- create deals with an attached photo for the businesses of which the user is registered as a business owner

## 2.2 Non-Functional

The non-functional requirements are the requirements that describe how the system should operate. The requirements are divided into requirements of the usability, scalability and robustness.

1. Ruby on Rails must be used for the implementation of the web service
2. HTTP must be used as the communication protocol
3. The iPhone app should not use abnormal battery
4. The web service should handle a maximum of $100,000$ end users and $10,000$ business users in total.
5. The web service should handle simple requests from at maximum $10,000$ end users and $1,000$ business users in a time scope of 1 minute.
6. The web service should scale horizontally
7. The database system should scale horizontally
8. The web service should handle photo uploads/downloads from at maximum 100 people in a time scope of 1 minute.
9. The web service should handle that every user uploads 1 image in average
10. A simple request that does not involve image transfer should at maximum take 500 *milliseconds*
11. It should not be possible to retrieve a member's password from the database
12. Transferring email and password in the network of the system should at maximum be transferred once per user session
13. The system should have automated tests

### 2.2.1 Justification of Non-Functional Requirements

The not obvious non-functional requirements are shortly explained.

Non-functional requirement 1: Ruby on Rails is known by the developers and because of time to market this is used. Non-functional requirement 4, 5, 6, 7 are because of the business that needs to reach out to numerous of people.

### 2.2.2 Scalability Requirement Analysis

The following sections analyses the requirements related to data transfer and storage. The overview of the requirements can be seen on table 2.1.

#### 2.2.2.1 Data Transfer

A proper quality image is estimated to be approximately 5 MB which is rounded off to $10^7$ bytes. In order to meet non-functional requirement 4 and 8 every transfer will be of $10^7$ bytes per end user resulting in approximately $10^7\ bytes/user \cdot 10^2\ user = 10^9\ bytes$. When counting with a time frame of 1 minute, this results in $10^9\ bytes/min = 10^7\ bytes/sec$.

#### 2.2.2.2 Data Storage

The data storage should be calculated in worst case, where the image uploads of $10^7$ is considered the largest transfer. The data transfer is estimated to be $10^7\ bytes/user$ in average. This will end up with a data storage of approximately $10^7\ bytes/user \cdot 10^5\ user = 10^{12}\ bytes = 1\ TB$ of data only for the images.

#### 2.2.2.3 Summary

Table 2.1 shows the worst case transfer and will be verified in the tests.

| **Parameter** | **Total** | **Live (in a minute)** |
|---|---|---|
| Amount of end-users | $10^5$ | $10^3$ |
| Amount of business-users | $10^3$ | $10^2$ |
| Response time | - | $O(1)$ |
| Data transfer per user | Worst case: $5 \cdot 10^7$ bytes/user<br>Average: $10^7$ bytes/user | Same |
| Data transfer rate | - | $10^7$ bytes/sec |
| Data storage per user | $10^7$ bytes per user | - |
| Data storage total | $1\ TB$ | - |

**Table 2.1:** The non-functional requirements related to scalability

Chapter 3

# Technology Decisions

This chapter is about why the different technologies were chosen in order to meet the requirements of the system. The architectural components are: client, web server, database and image delivery and are connected as illustrated on figure 3.1. This chapter is about the technology decisions behind these components and connections, which will be discussed and settled.

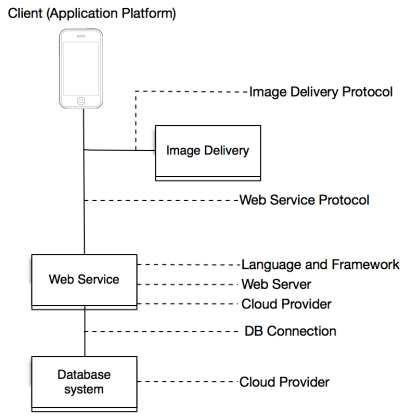



**Figure 3.1:** The component architecture

## 3.1 Scaling Strategies

In server architectures there are two strategies for scaling: horizontal and vertical. With horizontal scaling one adds more servers to the system, whereas with vertical one adds more resources (CPU, RAM etc.) to the existing server(s). The vertical scaling strategy is the easiest, but the scaling capacity is limited due to the fact that the servers in the cloud have maximum resource capacity. This means that vertical scaling only can be a solution up to a given point, where the demand hits the limit of the resources. The long-term solution is therefore to scale horizontally. Besides being able to scale, the horizontal scaling technique can also be used for solving robustness issues such as server failures. But even though horizontal scaling is superior in scalability and robustness it also adds complexity to the system in the form of balancing the traffic, connecting the system network and in the case of scaling database servers - data consistency.

Horizontal scaling is the long term scaling plan and with robustness advantages such as failure tolerance, the architecture will be designed upon scaling horizontally.

## 3.2 Technology of the Application Platform

As the iPhone is the platform, iOS will be used as the operating system. The decision for development whether to develop both for the Android and iPhone or select a development tool that does both, is what this section is about.

Objective-C with Cocoa Touch is used for the iPhone app development. A cross-compiler that translate e.g. C# to Objective-C does exist. A previous development experience with the cross compiler Mono Touch[1] showed difficulties. Mono Touch cross compiles C# to Objective-C. The application was complicated to debug because of the added unreadable layer. Performance was horrible even on small application. Only few animations and gestures existed.

Another way is to use frameworks such as, GabPhone[2] that creates apps using the web technologies HTML5, CSS, and JavaScript. The benefit of this is that with a few modifications an Android application can be created with the same code base. The drawback from this is that an Android app and an iPhone app have different design patterns and gestures. This means that the code base will

[1] `http://xamarin.com/monotouch`
[2] `http://phonegap.com/`

be separated or that it will have an influence on the user experience, which is a main focus in the project.

Writing in the native language for a platform is generally the safe choice even if that means learning a new language. When using cross compilers another layer of additional code is added. This has an impact on performance. This means that performance of an application written in the native language is higher. Another reason is the IDE for Objective-C - Xcode. The documentation, the storyboard and instruments are just some of the features that help the iPhone development. These features can only be obtained by using Objective-C with Xcode.

Higher performance, superior development tools, more animations and gestures, better user experience and overall an improved app experience for the user is preferred over faster development. Objective-C is chosen as the programming language of the application platform.

## 3.3 Image Delivery

In the requirements analysis in section 2.2.2.1, the worst case transfers happen when the users download images from the system, where the system ends up transferring 100 $MB/sec$. Besides the live data transfer, the data storage related to images also ends up being 1 TB of storage. The image storing and downloading is unrelated to the web service and could therefore be its own component.

That component should get its own IP or domain so that the image download will happen independently of the web service application server. A solution for this could be to use a database such as MongoDB. MongoDB has a technology called GridFS[3] that is optimized for storing files (up to 16 MB). To use MongoDB as a image storage component, a storage application should be implemented and deployed to its own server.

Instead of using time on implementing a file storage, the storage component could be outsourced to a Content Delivery Network (CDN). A CDN is a cloud service that provides data storage. The data storage is manipulated by an API e.g. by a Ruby Gem such as Fog[4]. Two candidates for a CDN in this project

[3] http://docs.mongodb.org/manual/core/gridfs/
[4] https://github.com/fog/fog

would be RackSpace's CloudFiles[5] and Amazon's CloudFront[6].

The core focus of this project and business is not to build custom components. The file storage will therefore be outsourced to a CDN provider, which will be RackSpace due to good experience in a previous project.

## 3.4 Web Service Technologies

The web service is responsible for serving the requests from the client and to query the database for the relevant data. The web service will be implemented with the object-oriented programming language Ruby together with the Rails framework. The following sections will consider the implementation of Ruby and the web server setup.

### 3.4.1 Ruby Implementation

At the moment there are three Ruby implementations ready for production that supports the Rails Framework: the official MRI (implemented in C), JRuby (implemented with Java) that runs on JVM and Rubinus (implemented in C++) that runs with the help from LLVM. The Rubinius implementation is not used by many members of Ruby community and is therefore not considered. The interesting decision is whether to use the MRI Ruby or JRuby.

The MRI Ruby is the official implementation and used by most Ruby programmers. The MRI Ruby implementation has a Global Intepreter Lock (GIL) that prevents the application from using threads. The MRI implementation is therefore not able to handle multiple request concurrently on runtime. [Kut13] This concurrency issue has been solved by application server frameworks such as Unicorn and Passenger that use processes instead to serve multiple concurrent connections.

The JRuby implementation is a Java implementation of Ruby that runs on the Java Virtual Machine. JVM runs threads that are more lightweight than processes, which means that the application can run more concurrent application instances and therefore serve more clients concurrently. The biggest disadvantages for JRuby is that many Ruby Gems are written in native C extensions, which are not supported yet and that the JVM has a slow start-up time [Ene].

[5] http://www.rackspace.com/cloud/files/
[6] http://aws.amazon.com/cloudfront/

| Implementation | Advantages |
|---|---|
| MRI Ruby | • Faster startup time<br>• Supports Ruby Gems written as native C extensions<br>• Thread-safe |
| JRuby | • Better garbage collection<br>• Can run more application instances per unit RAM<br>• Supports Java libraries<br>• Supports real threads in application |

**Table 3.1:** A comparison of the MRI and Java Ruby implementation

The comparison of the two implementations is shown on table 3.1. One of the requirements of this project is to scale and serve many clients, where JRuby seems to be best choice due to the fact that the JVM can serve more connections per instance. Although JRuby seems to be the best fit, the MRI Ruby was still chosen for the project, because none of the developers have had experience with JRuby and deploying web applications into the JVM. Syntactically the JRuby is the same as MRI, which means that a change to JRuby only will be a deployment issue. In further development, a change to the JRuby will be strongly considered.

### 3.4.2 Web Server Setup

The web server setup is important in order for the web server to be able to serve many clients concurrently. The web server setup consists of a web server that handles the network requests and an application server framework that extends the web server, such that the application is able to handle more requests concurrently.

The choice for web server was nginx, because the authors already know it. Alternatively Apache could have been as good, but has no major advantages that were favoured.

The application server is a layer that is able to extend the amount of connections to the web server by load balancing the application instances of the individual server instances. The choice of application server depends on whether one chooses the MRI or JRuby implementation.

The most used application servers that is supported by the MRI implementation are Thin, Unicorn, Puma and Phusion Passenger. [rt]. Puma has the advantage of running on threads, but it still can't break the GIL in the MRI implementation and is therefore better used for the JRuby implementation. There weren't found any proper benchmarks on the different application servers, so the decision to go for unicorn was due to a recommendation by one of the largest user of the MRI Ruby - GitHub [CW] and good experience in earlier projects.

If we had decided to use the JRuby implementation, it would have been possible to run on the JVM and thereby use the large Java Enterprise web servers such as Tomcat (by the JRuby Gem Trinidad) and JBoss (by the Ruby Gem TorqueBox). As explained earlier this would probably give a better performance due to the JVM's use of threads for request concurrency and will be considered in future development.

## 3.5 Web Service Protocol

The Ruby on Rails framework uses the resource-oriented architecture given by the REST architecture[7]. The REST architecture provides guidelines for designing an API that uses HTTP as it was designed such as using the HTTP headers and methods for requests and responds.

An alternative to REST would be the SOAP protocol. The SOAP protocol is a more strict protocol that comes with custom type checking schemas and several additions to improve security. The SOAP protocol could be implemented in Rails by RubyGems[8], but the advantages of SOAP is outweighted by the complexity that comes with the customization. The web service will therefore be implemented with REST.

## 3.6 Database System

The database component is responsible for all the data except for the image uploads that is handled by the image delivery component. Due to the fact that almost every requests results in one or more database queries, the database system also has requirements with relation to scalability. The big question related

[7]REST stands for Representational State Transfer. It was formulated by Roy Fielding in his dissertation "Architectural Styles and the Design of Network-based Software Architectures" from the year 2000

[8]RubyGems are the name for the open source Ruby libraries gems

to choosing a database technology is whether it should be a Relational Database Management System (RDBMS) or a NoSQL system. This section will introduce the two types of database systems, explain the different consistency levels and compare them with relation to scalability, consistency level, performance and productivity in order to find the database technology that fits this project the best. In order to fully understand the scalability properties for databases, the CAP Theorem is introduced at first.

The discussion will be theoretical on the natural designs of the different systems, because it is practically possible to modify and implement the systems such as one wants.

### 3.6.1 CAP Theorem

When the subject is scaling database systems; consistency, availability and partition tolerance are three important properties. We define these three properties of a database as:

**Consistency** A database system under consistency must have a total order on all operations.

**Availability** For a distributed system to be fully available every request must result in a response.

**Partition tolerance** Partition tolerance of a distributed system can be modelled by how tolerant the system is to lost messages between nodes in two different components.

These three properties are all important for a database. In an optimal database system the data is strongly consistent, the system is always available for reads and writes and a bad network should not cause the system to fail. Although they are all important properties, it has been proved that a database system only fully can have two out of the three properties as stated in Theorem 3 in a paper by Seth Gilbert and Nancy Lynch [GL02]. This prove is called the CAP Theorem[9]. It was originally proposed by Erik Brewer in 2000 [Bre00] until it was later formally proved by Seth Gilbert and Nancy Lynch.

By the fact that a database system only can choose two out of the three properties, database systems are split into three categories:

[9]CAP stands for Consistency, Availability and Partition tolerance

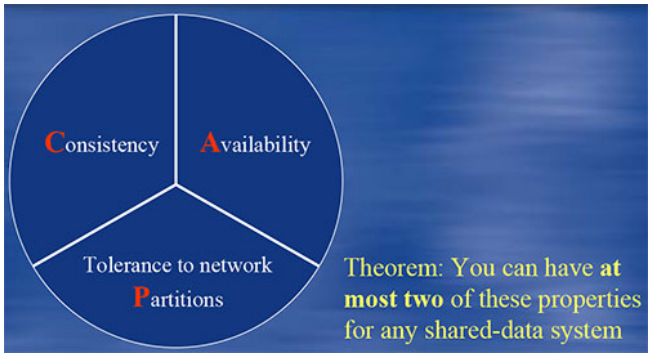



**Figure 3.2:** An illustration of the CAP theorem as presented by Eric A. Brewer in his keynote *Towards robust distributed systems* [Bre00].

- Availability and Consistency (AC)
- Partition tolerance and Availability (PA)
- Consistency and Partition tolerance (CP)

### 3.6.2 Relational Databases

The most used databases for the past few decades and now is the RDBMS's [DE13] that are based on the relational model. The relational model is the theory of having tuples with relations structured in tables to describe a data set. In order to query the data into the required form, the query language SQL is used to combine the relations with operations such as "selection" and "join".

Most RDBMS's implements the ACID transaction lock. ACID stands for Atomic, Consistent, Isolation and Durability. In overall the ACID lock makes sure that the database transactions are executed properly in order to preserve these properties. The ACID properties means that the database has immediate consistency (explained in section 3.6.4), but also that the transactions are locked in order to have an overall order.

RDBMS's are under the AC category of the CAP theorem due to the ACID implementation and the fact that the nodes are both available for reads and writes. Consequently RDBMS's do not have partition tolerance and are therefore not able to scale horizontally.

Within the Ruby on Rails community, the most used and mature with relation

to integrations of RDBMS's are *PostgreSQL* and *MySQL*.

### 3.6.3 NoSQL

In the need for more available and partition tolerant systems, a movement of non-relational databases that was ready to trade off consistency for other useful properties began in the beginning of the year 2000. The movement was lead by Google's BigTable [goo06] and Amazon's Dynamo [ama07] as the largest enterprises. This movement introduced a new category of non-relational databases called NoSQL[10].

Today several NoSQL database systems have been introduced with *MongoDB*, *Cassandra*, *CouchDB* and *Redis* as the most used NoSQL databases [DE13]. Martin Fowler and Pramod Sadalage divide the NoSQL databases into the categories: key-value, document, column-family, and graph. Except for the graph database, these categories share the characteristic of being *aggregate-oriented*. [SF12]

The tuple storage in relational database is limited to a single structure of values and does not support nested structures. An aggregate is a more complex structure that allows the insertion nested structures of records or lists [SF12]. Another difference is that the NoSQL databases are schema-less and are therefore unable to structure aggregates with prior to querying the data based on the attributes of the aggregates. The only NoSQL system that supports dynamic queries at the moment is the document-oriented MongoDB. Other NoSQL systems use database views with the map/reduce method.

The primary interest of the NoSQL databases is to run on clusters, which means that they favour partition tolerance. [SF12] From the CAP theorem, the NoSQL databases can either choose consistency or availability. This decision divides the NoSQL databases into two categories: the PA and CP databases.

Partition tolerance databases have the ability to scale horizontally, because the nodes in the cluster are able to work independently. The main scaling strategy is to replicate the databases across nodes so that they synchronize and at some point of time are all consistent. This is called eventual consistency (explained in section 3.6.4).

Although all NoSQL databases are equal in their primary interest of partition tolerance, the PA and CP databases differ by their cluster architecture, replica-

[10] NoSQL stands for "Not only SQL", which became the commercial name for these databases that not do not support the query language "SQL".

tion technique and consistency handling as explained in the following sections about PA and CP databases.

#### 3.6.3.1 Partition Tolerance and Availability

The PA databases have a peer-to-peer cluster architecture that consists of only master nodes, which means that every node are available for reads and writes as illustrated on figure 3.3. They all have equal priority and thereby equally correct version of the data, which means that data are not lost with node failures and it is easy to scale both for reads and writes.

The downside with the peer-to-peer architecture is the level of consistency. In the best case there is eventual consistency, but if two clients writes to the same record and hits two different nodes, there will be a write-write-conflict, which means that the two nodes have different versions of the same data set.

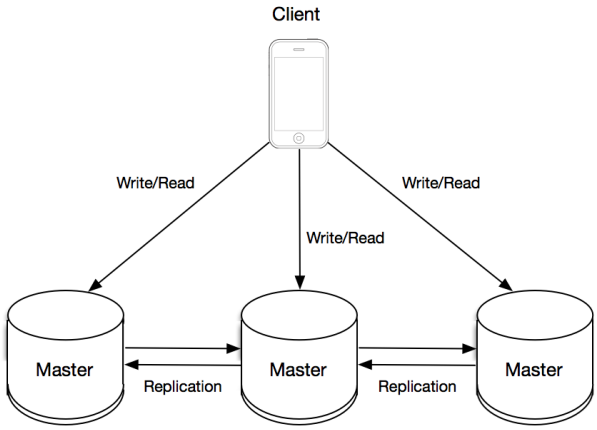



**Figure 3.3:** An illustration of the master-master cluster setup

#### 3.6.3.2 Consistency and Partition Tolerance

The CP databases have a master-slave cluster architecture that consists of only one master and more slaves. The one master is responsible for both reads and writes and the slaves are only responsible for reads as illustrated on figure 3.4. The master-slave architecture has its advantage in that all the writes have an order, which means that write-write-conflicts are avoided. The replication will

always come from the master, so there will never be a time, where the database is incorrect in the sense that there are conflicting data sets.

Although the master-slave architecture avoids incorrect data sets, it has a single point of failure in the master node. If the master node fails, the system will not be available for writes and will thereby loose data. This means that the database needs to have a failover strategy in order to have high availability. MongoDB handles master failures by prioritizing the nodes and when the master fails, the slave with the highest priority will be promoted to master. The failing master will then recover and become slave.

The single master can also be a bottleneck with relation to write traffic, because there only can be one master per data object. To solve this problem some CP databases supports the sharding method. Sharding the database means to structurally divide the database into more clusters. If the scaling was targeted on data about users, the sharding technique could divide the database into users with emails starting with A-M and N-Z.

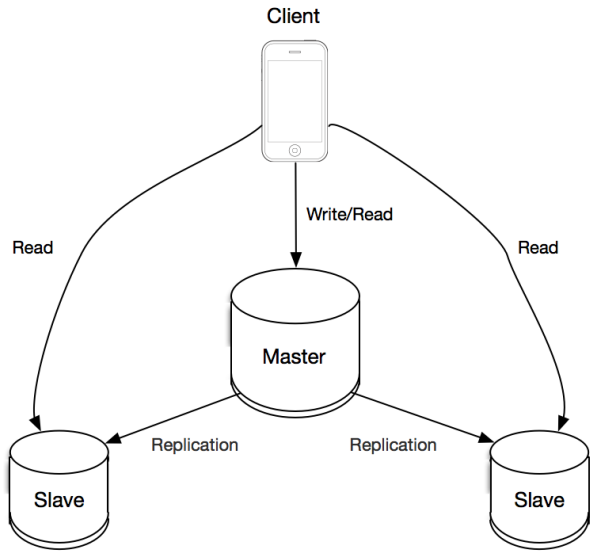



**Figure 3.4:** An illustration of the master-slave cluster setup

### 3.6.4 Eventual and Immediate Consistency

Optimally a database should always be immediate consistent, such that the data is the same for all users, but this is not the case for NoSQL databases. NoSQL databases are said to be eventually consistent. Shortly eventual consistency means that the different data sets in the different nodes will be consistent at some point of time. These levels of consistency are different for updates and reads.

An example that explains update consistency would be, if two people (let's call them David and Anders) share a business with a website. On that website the contact details are out of date, so David and Anders updates the contact details on the same time with different formats. The write from Anders accesses the database just before David's. In AC there will be a lock that prevents David's write from being executed, because David actually corrected an old version of the data and not the version from Anders. In PA databases there would be a problem if Anders' change hit one database and David's hit another. This would mean that, when the database replicates to each other, there would be a merge (write-write) conflict. In CP databases this conflict is avoided, because there only exists one master that keeps an order of the transactions. Instead the default database setup would result in David's changes overwriting Anders'.

Now let's say David and Anders are to book a hotel room. David is in San Francisco and Anders is in Denmark. In the meantime while they discuss which hotel to choose over the telephone a third person Johan from Belgium reserves the last room available at Wonder Hotel. Right after Johan's booking, David and Anders looks at the last room from Wonder Hotel on the website. If the database was a AC database, they would both see that the room is booked, because their read came after Johan's booking. If the database was a PA or CP database, there could be a different case: they both see the Wonder Hotel page, where on Anders' computer it is booked and on David's there is still a room left. This is called replication consistency or an inconsistent read, because the nodes haven't replicated fully such that some nodes doesn't have the latest version of the data (as with the database server that Anders hit). This situation is illustrated on figure 3.5.

### 3.6.5 RDMBS vs. NoSQL

The database system will be discussed by the categories given by the CAP theorem: AC, PA and CP. The three major parameters that these three database systems are evaluated upon are: querying features, data modelling, scalability

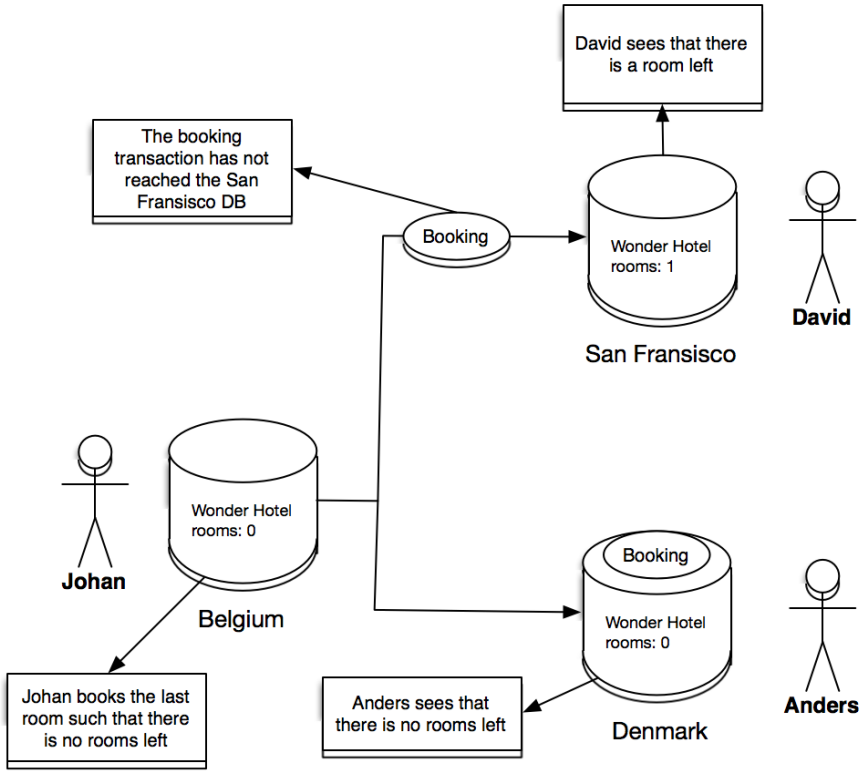



**Figure 3.5:** An example of an inconsistent read, where the hotel booking transaction has not been synchronized to all nodes. Example inspired by Martin Fowler and Pramod Sadalage [SF12].

and consistency level. Performance would also be a qualified parameter, but the performance is dependent upon the use case, which makes it hard to relate to this project. These parameters are all important, but it is also important to meet the requirement regarding time to market. A comparison overview of the systems is shown on table 3.2.

The RDBMS's that implements ACID doesn't have the ability to scale horizontally, which means that scaling RDBMS's by its nature has limits. Also with relation to the functional requirements, the eventual consistency is not a problem. The NoSQL systems are already new with relation to querying and modelling the data, so from these databases, the ones that is most similar to the well known RDBMS's is favoured. From the three categories, Document-

Oriented Modelling (DOM) is the most similar due to fact that it is only semi schema less and has a flexible structure, where the others have almost none. With MongoDB it is also possible to have dynamic queries, which is a more flexible and easy way to retrieve data compared to the map/reduce method. From these considerations the RDBMS is dropped due to lack of horizontal scalability strategies, which leaves the decision between the document-oriented databases MongoDB and CouchDB. CouchDB is a PA database with the possibility to scale writes, which could be a requirement do to the fact that all users are able to post content to the system such as comments, ratings and photos. Even though CouchDB has an advantage with scaling writes there is at the moment no support for dynamic queries. Here MongoDB is chosen because of the support for dynamic queries, which will give a more flexible and well-known way of developing the application. If CouchDB implements dynamic queries as well as MongoDB in the future, it would probably be considered to move to CouchDB.

| | **RDBMS** | **NoSQL** | |
|---|---|---|---|
| | **AC** | **CP** | **PA** |
| **Querying** | Universal query language (SQL). | MongoDB supports dynamic queries. The rest uses map/reduce. | Map/reduce. |
| **Data modelling** | Relational Data Modelling | • Document-Oriented (MongoDB)<br>• Column-Oriented (HBase, BigTable)<br>• Key-Value (Redis, MemecacheDB) | • Document-Oriented (CouchDB)<br>• Column-Oriented (Cassandra)<br>• Key-Value (Riak, Dynamo) |
| **Write scaling technique** | Only vertical (with ACID). | Some DB's have support for sharding (MongoDB and Redis) (horizontal) else vertical. | Adding more nodes and replication (horizontal) or vertical. |
| **Read scaling technique** | Only vertical (with ACID). | Adding more slaves and replication (horizontal) or vertical. | Adding more nodes and replication (horizontal) or vertical. |
| **Consistency level** | Immediate consistency (with ACID) | Eventual consistency, but avoids write-write-conflicts | Eventual consistency. Write-write-conflicts should be handles in the application. |

**Table 3.2:** Comparison of database technologies divided into the choice made with relation to the CAP Theorem (AC vs. CP vs. PA)

## 3.7 Database Connection

In order for the web service to query data, there need to be a connection to the database. For a single server setup, a native database setup would be possible by installing the database on the application server and use a database driver given by a Ruby Gem.

If the web service and/or the database system should be able to scale horizontally, the databases would require their own nodes. With a seperated database system, a network connection is required. The connection to a seperate database could be implemented with HTTP or TCP, but further research has not been a part of the project.

A solution would also be to use a Database as a Service (DaaS) and outsource the system. This way it is just a matter of configuration on the application platform and the database system will be available and easy to scale

In the development phase the connection will be a native implementation, because it is a simple setup. When the system goes into production, a DaaS will be strongly considered, which means that the database connection will be a matter of configuration in the web service.

## 3.8 Cloud Provider

The service provider that offers storage and the service application can either be through a private or a public cloud. In a private cloud one owns the servers in a data center and has fully control, but it also means fully control of server maintenance. With a public cloud (cloud from now on) solution another company takes care of this maintenance. A reason for choosing a cloud provider is scalability. When needed scaling a private data center would be a too large investment in the beginning and network administrators are needed. The main reason for not having own data center is the lack of skills and experience with server maintenance which either of the developers have. Experienced cloud providers like Amazon or RackSpace would be a better choice for this start-up. Many different cloud providers exist, but only three different services and the most relevant for this project will be covered here.

### 3.8.1 Cloud.dk

Cloud.dk is the first Danish cloud provider and went live in 2011. They have their data center is in Denmark. [weba] This could give a small performance boost because of the closer location compared to Amazon in Ireland, but how big the performance boost is has not been tested. Cloud.dk was chosen in the development phase because of price and previous experience with deployment.

### 3.8.2 Amazon Web Services

Amazon Web Services (AWS) provides a Infrastructure as a Service (IaaS). AWS have several solutions but only Amazon Elastic Compute Cloud (EC2) is compared because of the best scaling capability. AWS has features such as Auto Scaling and CloudWatch, which monitors each instance and checks if less or more instances are needed. CloudWatch tells Auto Scaling that a server is running hot and starts a new instance, but also if an instance is running cold it stops it in order to lower the costs. An Auto Scaling policy can be set-up e.g. when and to which group a new instance should start. AWS also provides a load balancer that directs all the incoming requests and decides which instances the request should be handled by. These features are needed in production of this system, and Cloud.dk does not have services like these. Also availability is another reason for using AWS. The EC2 commitment is 99.95% availability for each EC2 Region [webb]. At the moment Cloud.dk does not say anything about their availability.

### 3.8.3 Heroku

Heroku is the most expensive compared to Cloud.dk and AWS, but delivers a high-end service. Heroku uses AWS as their cloud provider and provides Platform as a Service (PaaS), where deployment, load balancing and network setup is taken care of together with tools for one-line-command scaling. On the other hand you have no control over how the infrastructure of your deployment, which could be a problem if advanced set-ups are a requirement. [webc]

### 3.8.4 Decision

At this moment Cloud.dk is chosen because of ease and expenses, but because Cloud.dk do not provide easy scaling solutions and has been in release for less

than 2 years, another cloud provider is needed, when the system goes into production.

Comparing an IaaS (EC2) with a PaaS (Heroku) is difficult, but in the end it all comes down to cost versus lost hours of implementation. Using EC2 requires a system administrator when scaling and Heroku is more expensive. This project is a start-up where the priority is on product development not advanced infrastructres. For future development Heroku is in favor, because the extra cost per instance seems cheaper than the time spent maintenance and debugging deployment recipes.

## 3.9 Conclusion

In order to meet scalability requirements in the long term and to be more failure tolerant, the architecture is designed upon scaling horizontally. This sets requirements for the technologies chosen for the project. The web service will be written in the object-oriented language Ruby with the framework Rails. For developing the iPhone client App, the native programming language of the iPhone, Objective-C is chosen together with the IDE Xcode due to advantages related to better performance and better integration with the CocoaTouch API's.

The database system was chosen to be in the category of NoSQL databases. If a database system needs to be partitioned as with horizontal scaling, the trade off needs to be the consistency or availability due to the CAP theorem. Relational databases are able to do this, but NoSQL has it built into its nature. The database system chosen is MongoDB.

Choosing a provider for the service and storage is about control against time spent on servers. A cloud provider is chosen against a private because server maintenance is not wanted. Cloud.dk is chosen as cloud provider, but will be replaced with either Amazon EC2 or Heroku later due to the fact that more implementation hours are needed to scale with Cloud.dk. Whether EC2 or Heroku should be the cloud provider is a choice between more features (Heroku) or more resources and control (EC2). At this moment Heroku is in favor when launching.

The requirements of both the end users and business user to be able to upload photos means that a lot of traffic is related to downloading and storing images. The image storage part is therefore seperated into its own component hosted by RackSpace.

The final architecture diagram of the technologies chosen is visualised in figure 3.6.

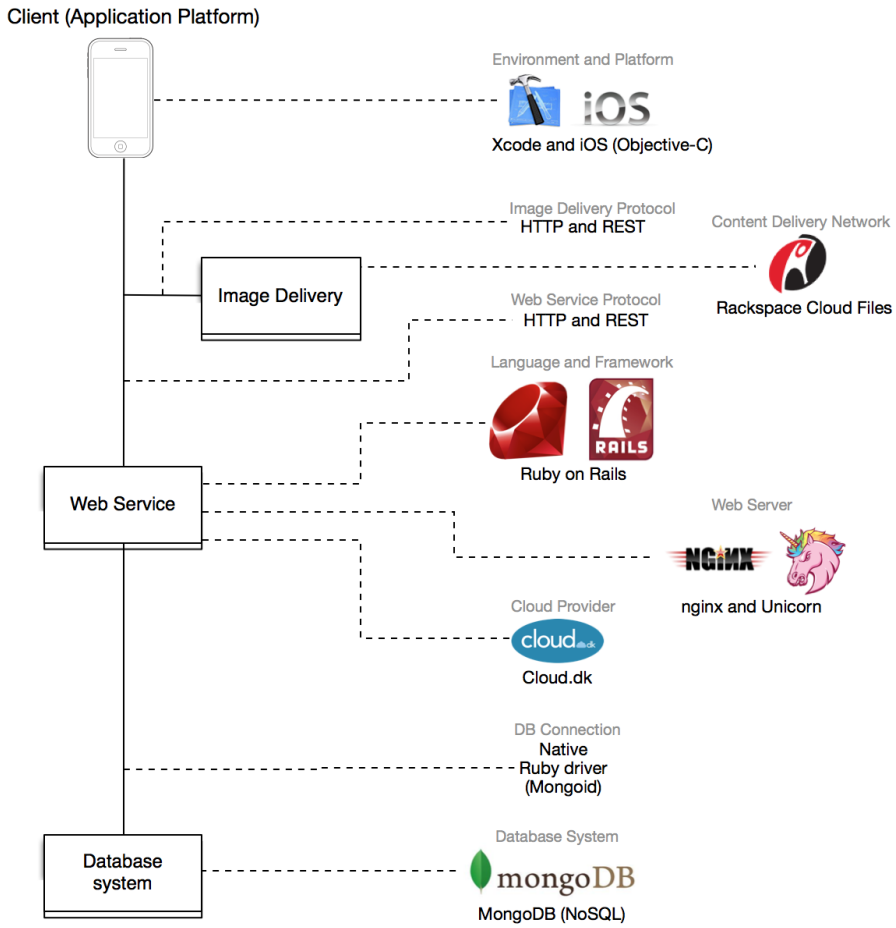



**Figure 3.6:** The architectural components with the technologies chosen for the project

Chapter 4

# Technical report

With the decisions of the technology settled, the analysis, the design and the implementation of the system is what the following chapter is about. The results of the tests performed on the system is in the end of this chapter. The aim of the this chapter is to describe and outline the thoughts on how the product was developed.

## 4.1 Analysis

This chapter is an analysis of the requirements with a view to predict the challenges related to the requirements. With the requirements as foundation of what the system should consist of, this chapter contains an analysis of the robustness in the system, the data needed about the locations, the user experience of the iPhone app and modelling data for document-oriented databases.

### 4.1.1 System Robustness

Robustness in a software program refers to the program performs well not only under ordinary conditions, but also under unusual conditions that stress the de-

velopers assumptions. The chapter will discuss the robustness aspects; security, availability and error handling of the system.

#### 4.1.1.1 Security

Even though the majority of the users does not want to damage or hack the system, there is still a possibility that some users have this intention. In order to be able to handle these users, the developer must be in a state of paranoia[1] and do not trust the users. The degree of paranoia should be adjusted to the potential damage that could be caused.

There are two main reasons for security in the system:

1. Protect private information from the users (such as email and password)
2. Protect the identity of the users

The security protocol of the system needs to solve both problems in order to be protected by attackers.

On an unsecure Wi-Fi network, it is possible for an attacker to sniff up the network traffic from other users that are on the same unsecure network. If the traffic contains private information such as login credentials, the attacker is able to obtain this information. A well-known method for eliminating the need to send private information or credentials over the network is to store the private information in a database and let the client authorize with a session key often in form of a hash key. All though this will prevent the system from transferring private information, it will still be possible for the attacker to sniff the session key and use it to authorize into the system on the behalf of the original session key owner. This is called session hijacking. The network sniffing is illustrated in figure 4.1.

Besides being able to sniff the traffic off the users that are on a mutual unsecure network, there is also the possibility for the attacker to hack the database in some way or to hack the process in which the user get to know his forgotten password. The data in the database and in the resources of the web service should therefore also be hidden for the attacker.

[1] One of the principles of robust programming described in [BF04]

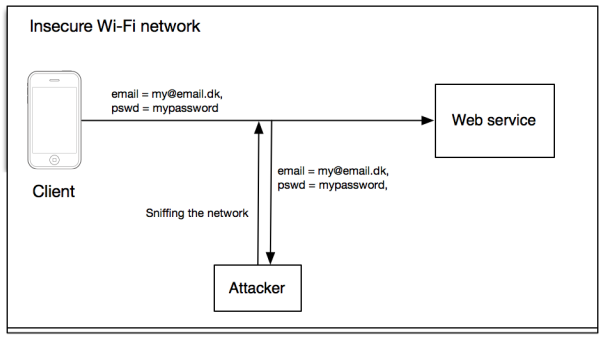



**Figure 4.1:** How a network sniffer acts: The network sniffer is able to read the request and retreive the user's credentials

This means that in order for the system to be secure, the developer should be in a state of paranoia due to the fact that an attacker is able to sniff the HTTP-traffic and possibly hack the database as seen in many cases.

#### 4.1.1.2 Availability

When the servers are hit by failure, the client is the victim in the sense that the user experience of the client relies on the availability of the system. The availability of the system can be judged by the availability of the different components used in a request from the client, the iPhone in this case. In overall the different components of this system is illustrated on figure 3.6. It is therefore important that the full system consisting of the web service, database system and the CDN are fully available. In order for a system to be available, there cannot be a single point of failure. A typical strategy for avoiding this is to at least have two nodes for every component.

The CDN is outsourced, which means that it is the responsibility of the provider to secure availability of that component. With the web service, the availability relies on the network implementation and deployment. With Cloud.dk as the provider, the availability relies on the deployment, where if the provider were a PaaS as Heroku it would be secured by the availability commitment of the provider. For the database system, the availability also relies on the network implementation or cloud service, but the database system itself needs to be available. As MongoDB is chosen as database there can't be fully availability, because MongoDB is a CP database. But as explained section 3.6.3.2 MongoDB has a failover strategy if the master node fails. This means that full availablity

cannot be ensured, but high availability can.

#### 4.1.1.3 Error Handling

Even though the optimal system would consist of an iPhone App with no bugs and a fully available system, it is a fact that developers are humans and make mistakes. The situations that should not happen also need to be handled. Users need to be informed about the problem and given sufficient information on how to solve it (if they are able to). The developers need detailed reports of the client, state of the system and stack trace in order to be able to find the bug. For exceptions and crashes there exists services such as Google Analytics iOS SDK[2] for the iPhone and Airbrake[3] for the web service. The users need information by explainable messages shown in the UI.

### 4.1.2 Location Data

An application with no data (also known as a cold start) is not a useful experience for the first users. Data about shops are needed, because the user should be able to locate shops that have not been registered on the iPhone app. The two qualified location data providers are currently Open Street Maps (OSM) and Google Places. OSM is a collaborative project, which means that everyone can create and edit in the map.

The OSM open source map gives more flexibility e.g. you can create locations to the database without any verification. In Google Places this is a process that needs verification from the Google company. On the other hand by testing some few queries, such as "pizza copenhagen" and "coffee copenhagen" on OSM and Google Places it seemed that Google Places have more data in Denmark.

Google Places have some useful features such as autocomplete and text search that means when typing "coffee", Google Places will translate it to "cafes" and thereby getting a better outcome in the search. Google Places search seems more intelligent which gives a better usability for the user who is on the run.

The biggest advantage for choosing Google Places against OSM is the Google Places services. With OSM the database file of 28GB, which contains the whole world had to be integrated with the database together with implemented search

[2] https://developers.google.com/analytics/devguides/collection/ios/
[3] https://airbrake.io

algorithms. This solution would give more control, more flexibility and a possibility to optimize the location algorithms to our needs. Choosing the Google Places API means that the system has a dependency that is not controllable, but the API is reliable and has had great performance in our experience.

Google Places API only allows 1000 request pr. day, but by registering with a credit card 100000 request pr. day is allowed. If wanted even more requests one have to contact and make an arrangement with Google. Changing to OSM in the service would not have an effect on the user when the iPhone app is on the App Store. This would perhaps be necessary if an arrangement with Google is not possible and more than 100000 requests are needed.

| | OSM | Google Places |
|---|---|---|
| More flexibility | X | |
| No request limit | X | |
| Better search results | | X |
| More data | | X |
| Less implementation | | X |

**Table 4.1:** A comparison of OSM and Google Places as location data providers for the project.

A comparison table of the two providers can be seen on table 4.1.In the end Google Places was chosen because of better user experience, more data and fewer hours of development.

### 4.1.3 User Experience for the iPhone App

When using a software application the experience with the software is called the user experience (UX). The one reason for giving the users a good experience is happy customers. Happy customers are loyal customers that refer the product over others.

User experience for a software application is subjective. The user experience of a program involves the emotions of the person using it. The best way to get great user experience is by asking the customers and then listen, observe and notice. But because this is a unique product a first prototype is needed. This prototype is the first impression that is the reason why obvious UX mistake should be avoided.

The book Mobile Usability by Jakob Nielsen and Raluca Budie [JN13] has techniques backed up by user tests in order to statistically prove their work. Many of these techniques have been used in this project in order to have user experience that is based on user tests and statistics instead of intuition. The techniques and the related the value to the users are shown in table 4.2. Only the techniques that have been used in this application is described. In the design these techniques are applied to the iPhone app.

| **Technic** | **Description** | **Value** |
|---|---|---|
| Early registration must die | Avoid the first screen to be a registration screen | • The new user can see what the app provides instantly without time spent or effort used.<br>• Motivates the user to use it.<br>• The user is more likely to tell friends about it.<br>• More downloads |
| Minimize interaction cost | Interaction cost refers to the number of atomic actions that a user needs to perform to fulfill a task. | • Faster use of the app<br>• Fewer downloads mean less waiting |
| Less typing on mobile | Most users hates typing on a virtual keyboard. | • Minimize interaction cost |
| Bigger touch targets | Research indicates a target size of 1 cm by 1 cm is ideal. | • Less mistakes mean fewer interactions. |
| Two interfaces for same data | Showing the same data with different interfaces can be useful e.g a map and a table view for the same data. | • Gives the user overview and details |
| Don't overload | A screen on a mobile devices is small do overload it. | • Better overview of information |

**Table 4.2:** User Experience techniques by Jakob Nielsen and Raluca Budie and the value that they give to users.

### 4.1.4 Document-Oriented Modelling

Data modelling in relational databases are often driven by normalizing data in order to eliminate update anomalies. A document in a document-oriented database has no database schema or types, which requires new guidelines for modelling the database. Two main considerations in DOM is as defined by the MongoDB documentation [10g13]:

- How the data will grow over time
- The kind of queries your application(s) will perform

These considerations can lead to the decisions of modelling methods such as:

- Normalization vs. denormalization
- Embedded documents vs. referencing documents
- The index strategy

In these considerations strong consistency and normalized data is the trade off in favor of better performance.

#### 4.1.4.1 Eventual Consistency

The problem with eventual consistency is as explained in section 3.6.4. In the web service of this project, there is not important matters where the eventual consistency is a problem are edge cases. Even though these edge cases ends up in a complaint, they happen so few times that the damage can be handled. The authors behind this project adds the following guideline to DOM:

> *The considerations bound to data inconsistency should be compared to the potential damage to the company behind the application.*

#### 4.1.4.2 Data Denormalization

It is not a good practice to have update anomalies in software. If e.g. data consistency relies on the application level, it is a great source for bugs. But

with DOM it is sometimes necessary when queries in the application need to perform well. This introduces a data modelling design decision called data denormalization. Denormalizing the data means to have the same data twice or more in the same database. This method is especially used for polymorphic data. An example would be the entity of a comment. A comment is often the same no matter what the comment is for. In relational modelling this would be a good case of a polymorphic table for the entity "Comment". In DOM the concept of joins is avoided due to the focus on availability and partition tolerance. The aggregation of comments relies on the queries made in the application. If the focus were on queries executed on comments in general, it would call for a comment aggregate, where the link between a comment and the commentable entity is made by a list of references in the commentable entity. If the focus on the other hand were queries made on the commentable entities, it would be best to embed the comments in the aggregate of the commentable entity.

In the last solution of the comment data, queries on all comments across the different commentable entities would not be possible. If such queries were important for the application, denormalization would probably be the solution. Denormalization in this case would mean that every time a comment is added, it is created as a part of the commentable aggregate and thereby in its own comment collection. This method adds complexity to the web service due to the fact that every time a client edits or deletes a comment, the application would need to execute the command in two collections. If this complexity isn't handled across the web service, the consistency level of the data will change from eventual to never.

## 4.2 Design

The predicted challenges are now taken into account. This chapter describes and discuss the difference design options for the system with use of the chosen technologies. The focus of this chapter is the thoughts in general and not specific implementation solutions. This chapter contain design discussions about the robustness, iPhone app, location data from Google Places, web service and database system.

### 4.2.1 System Robustness

The whole system is designed for robustness in order to have a system that continuously can deliver the best user experience to the user of the application.

This section will explain the design decisions behind the robustness of the system. This includes robustness subjects such as main tenancy, system testing, availability and security.

#### 4.2.1.1 Maintenance

Software maintenance is closely related to the robustness of a program. If the developer finds it easy to change and add features then it is often a sign of robust software. The code must be understandable, testable and readable which is difficult. Before the developer can change code he must understand what it already does. In order to meet the function requirements in section 2.1 many features had to be implemented. A clean structure of the code base can save time when further features are needed. In this project three different kinds of methods have been used in order to have maintainable code such as code reviews, software architecture and automated testing. Automated testing will be explained in section 4.2.1.2, but this section will go through code reviews and software architecture.

**4.2.1.1.1 Code Reviews** With code reviews, the code has to be accepted before merged into the main code base (read more about the process 6.2). Code review was used for improving the overall quality of the software, the developers' skills and in order to make the code maintainable. A developer that did not write the code should be able to easily understand the code, otherwise the code needs structure and a rewrite. In the meantime both developers should know what each method does.

**4.2.1.1.2 Software Architecture** The software architecture of the applications is important in order to keep the level of productivity, find bugs and make changes in the system. One of the parameters that help doing these things is the control of dependencies. The dependencies in the code base prevent changes due to the fact that a developer has a mind-set of "what happens if I change this code". If the dependencies are under control and minimized, the developer will easier be able to understand the system, detect bugs and change implementations.

The software architecture of the iPhone App is chosen to be Model-View-Controller, which will be explained more deeply in section 4.2.2.4.

The software architecture of the web service is MVC as well, which is the architecture dictated by the Rails framework.

| Component | Testing method | Prioritization |
|---|---|---|
| Web service | Unit-testing | On all objects |
| | Controller testing | On all controllers. Tested isolated. |
| | Integration testing | Only for important/risky integrations |
| | Profiling | On large algoritmic tasks. Are first made after implementation and therefore not in this project. |
| iPhone App | Unit-testing | On all model objects |
| | Controller testing | None due to the complexity of controllers |
| | UI Testing | On important UI flows |

**Table 4.3:** The test methods considered and their priority

#### 4.2.1.2 Automated Testing

Manual software testing is performed by a human sitting in front of the software and carefully going through the screens of the application. With automated testing pre-recorded and predefined actions compares the results to the expected behaviour and reports whether it succeed or failed. Automated testing helps the developer make changes in the system without testing the full system over and over again. By having a test suite, the developer can make the changes, run the test suite and thereby be verified whether or not part of the system has broken due to the changes.

The two components in the system that should be tested are the web service and iPhone App. Both components support several testing methods. Unit testing, controller testing, integration testing and profiling were the testing methods considered for the web service. For the iPhone App there were: unit testing, controller testing and UI testing.

The different tests take time to make and maintain, and they have therefore been prioritized with relation which situations they are relevant. These prioritizations are listed on table 4.3. The detailed explanations of how the different methods are used is explained in the implementation section 4.3.3.

### 4.2.2 iPhone App

The analysis of the user experience is described. This section is the design of the iPhone app and the features in order to meet the functional requirements.

#### 4.2.2.1 Features

The iPhone features needed for the minimum viable product and what values they bring to the app. Sorted by importance.

| **Feature** | **Data shown** | **Value to end user** | **Value to business user** |
|---|---|---|---|
| List view of deals | • Nearby deal list<br>• Deal photo<br>• Shop name<br>• Price<br>• Rating<br>• Time left<br>• Short description | Attention on current issued deals where the product photo is in focus. | Their product is in focus. |
| Map view of deals | • Map with nearby deals<br>• Description<br>• Small photo | Focus on where the deals are | |
| Detailed view of a deal | • Specific deal<br>• Deal photo<br>• Shop name<br>• Price<br>• Rating<br>• Time left<br>• Short description<br>• Longer description<br>• Number of comments | More information about the deal | |
| Member login | | Better ratings and comments, because a user is only allowed to rate once and authentication gives better comments | Find specific customers |

| | | | |
|---|---|---|---|
| Rating | | Possiblity to give opinion. Indication on quaility of deal or place | Figure out what customers want |
| Comments | • List of comments<br>• Comment photo<br>• Comment text | Possiblity to give more detailed opinion. Authenticate deal by comparing the deal photo and comment photos | Gives understanding of their business |
| Business registration | | | Correct name and address from Google Places API. Possibility to create deals |
| Business dashboard | • Old deals<br>• Active deals<br>• Future deals | | Overview of the shop deals |
| Creating deals | • Description input<br>• Price input<br>• Before input<br>• Take photo<br>• Add start time<br>• Add end time | | Increased sale by showing product to customers. In less than a minute a business user are able to create a deal. |
| Places | • 20 most relevant shops nearby showed in locations and list<br>• Shop name<br>• Shop category<br>• Shop rating | Gives the user a fully overview of both where and what is nearby. Only the most useful information is shown. Can access full map view by taping on map. | |

| | | | |
|---|---|---|---|
| Detailed place | • Address<br>• Shop picture<br>• Amount of comments<br>• Rating<br>• Description | Additional information of selected shop where ratings and comments are located. | Posibility to add more useful information about their shop |
| Search in places | • 20 most relevant nearby shops based on the search text both showned on map and on list<br>• Shop name<br>• Shop category<br>• Shop rating | Specific strings such as “pizza” or “coffee near Lyngby” can be searched. Gives the user a quick way of finding specific shops | |
| Updating with other coordinates | • 20 most relevant shops in the area of the map view based on the search text showed in locations and list<br>• Shop name<br>• Shop category<br>• Shop rating | Gives the user a posiblity to search other places than nearby also with a given search input. | |
| Inspire me | • Random deal | Takes a decision for the user. | |
| Live | • Latest rating and comments | Gives the user a sence of what is currently happing nearby. | |
| | | | |

The features listed in table 4.4 gives the end user a tool where he can find the exact shop he wanted and a quality deal. The business user who owns a shop can in less than a minute reach out to customers with a specific deal.

#### 4.2.2.2 Feature Flow

To get a overview of the features and how they are reached a flow diagram of the features is shown 4.2.

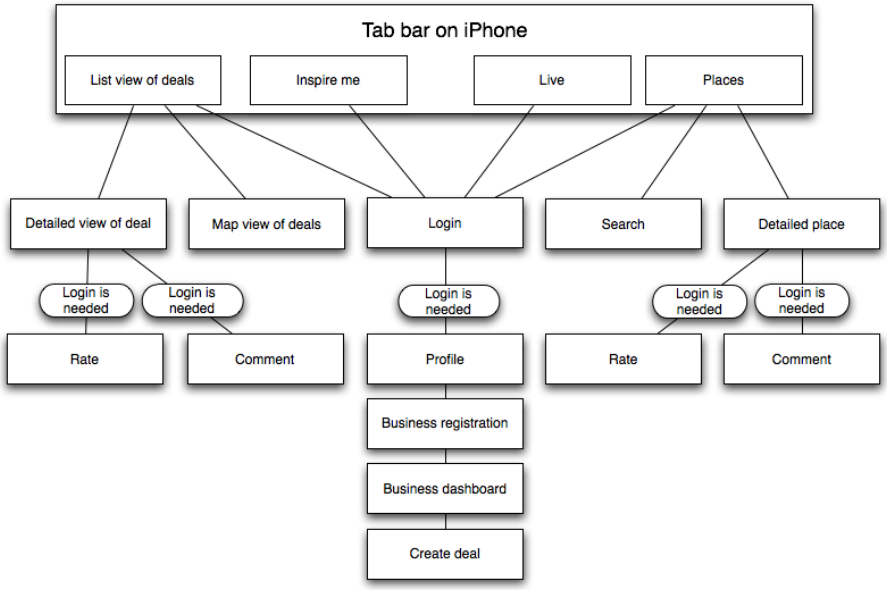



**Figure 4.2:** Feature flow shows that the login button is able to access from every tab.

As seen in the feature flow 4.2 from every tab button the login button is accessible placed in the upper right corner.

#### 4.2.2.3 Applied User Experience

The most useful user experience techniques in order to give the user a good experience when using the app have been analysed. Now it is up to applied these techniques to the application.

**Early registration must die** is used so the user is introduces immediately to the nearest deals. This is the core of our product and the user is not forced to register. Other techniques such as the user is not allow to see the detailed deal before logged in can be use to get more registered users.

**Minimize interaction cost** with only 1 interaction information is given. As soon as the user tabs on Places an update of the nearest shops are shown. Only when creating a deal a deeper layer exist and 3 interactions are needed. Hopefully more end users than business users will use the app and it would distract an end user if e.g. a tab in the tab bar was only for business users.

**Less typing on mobile** is shown when a business user creates a deal. Only the most important information about a deal is needed for a business user. When selecting start and end time a date picker is used instead of using the virtual keyboard. Which is the same reason for a slider when rating. Before launching when searching in Places some predefined categories should appear. Also when the business user registers their shop, a search based on location is preferred.

**Bigger touch targets** every button and touch area is 1 cm by 1 cm or larger. Only the annotations in the maps are slightly smaller, but this is chosen so they did not fill out the map.

**Two interfaces for same data** where shops are shown, both a map and a list view show the data.

**Don't overload** only the most important information is given to the user when seeing the deals. And only the description and a picture of the deal is shown when pressed on annotation in the map.

It the end this just give the user a fast and a good experience when using the app.

#### 4.2.2.4 Software Architecture of the iPhone App

The software architecture helps the developers keeping structure of the code and get a control of the dependencies in the system. As mentioned in section 4.2.1.1.2, the software architecture of the iPhone App is chosen to be MVC. This section will explain in more details about MVC architecture followed by how the MVC is used in this project.

**4.2.2.4.1 Model-View-Controller** MVC is a software architecture which is used in the iPhone app. MVC was chosen for separation of models and the user interface which is useful for encapsulating data and code reuse. Another advantage of MVC, in this project, is that the models can be reused when making an iPad app and if a newer version of iOS has different design patterns.

**Model** represents the data resources. The models take care of the data in the iPhone app and it is where the business logic exists. When getting request from the web service the model maps the JSON objects to the right variables.

**View** represents the user interface. A view can have an action, such as a button, which sends a notification to the target on the controller, and tells that this button has been pressed.

**Controller** handles the communication between the models and the views. Every view in the storyboard has a controller that controls the events happening in the view.

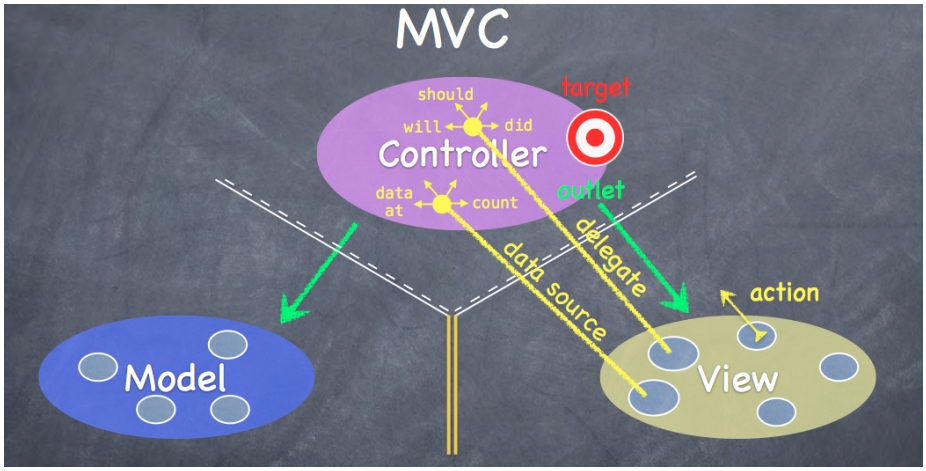



**Figure 4.3:** MVC for the iPhone app. Slide from iPad and iPhone Application Development September 27, 2011, Stanford University CS193p

As shown in the figure 4.3, there is a one-direction communication between the controller and the model and the controller and the view. The model and the view are not allowed to communicate and will not know anything about the controller.

**4.2.2.4.2 Move Code into Models** When using MVC in the iPhone app a useful automated test of the controllers is difficult to implement. Automated tests on controllers in the iPhone app were not practiced. The problem with this is that when a developer wants to change the code base and accidently deletes a view to features, an automated test does not fail and tells the developer this. This is difficult to avoid, but what the developer is able to do is to have as much

code in the models as possible. Automated tests of the models are much easier and do not change as repeatedly. By writing as much of the functionality as possible in the models the code is more maintainable.

### 4.2.3 Location Data

When the iPhone user needs to find nearby stores or specific stores within a category the Google Place API is called. Every request from the iPhone is sent to the service, which handles the request and sends it to the Google Places API. The response from the Google Place API is then handled in the service, which sends the correct response back. The reason for sending it through the service and not directly from the iPhone to the Google Place API is for the system to be able to control the data (and merging it with our own). This will have influence on performances due the extra HTTP request through the web service layer, but when the iPhone app is launched, the only way to change the product is through the service or with an app update which usually takes up to two weeks before getting to the App Store. In order to have more control with the product, when the iPhone app has reached the App Store, this is chosen instead of the higher performance.

Google Places have some defined types for each place, such as “bakery” and “cemetery”, but in this project only stores with retail is wanted. Types like “cemetery” and “bus station” are sorted out due to this are not what is wanted for this product. Additionally fewer overall types are replaces with custom categories such as “Nightlife” and “Arts and Entertainment”.

### 4.2.4 Web Service

This section is about the design of the web service - how the data is modelled with a document-oriented database, how the web service exposes resources as a resource-oriented API and in the end a short documentation of the API.

#### 4.2.4.1 Data Model

The database system used for the web service is the document-oriented database MongoDB. Document-oriented modelling is designed with focus on how the data will grow in time and the kind of queries made in the application as stated in section 4.1.4. The application in this case is the iPhone App, but other platforms

will be introduced in future development as described in the problem description section 1.1. In order to reduce the level of integrations to be maintained, the web service should optimally be designed for the general purpose and not for a specific platform. These two design rules are contradicting and they can therefore both not be fulfilled. The purpose of this project is to develop a system for the iPhone and the design rule of building a general API is therefore traded off for building a specialized web service for the iPhone App.

In the following sections the data within the features of the app are analyzed upon modelling the database.

**4.2.4.1.1 Deals and Businesses** The features of the iPhone app and the related data has been introduced in section 4.2.2.1. In this table it can be observed that the different views displays different data. From this table, the deal list view and deal map view displays data from deals and the places tab displays data from businesses (and the Google Places API). These aggregates deal and business are both filtered and sorted. This is an indication that there should exist a *Deal* and a *Business* collection. Although the data roughly fits these aggregates, the name and location of the business of a given deal is shown in both the deal list view and deal map view. If this data should be normalized, the deal should fetch a whole business document every time a deal list should be shown. This would indicate a bad document-oriented design and results in a $n + 1$ query and probably bad performance. The solution to this problem is *denormalization*. In this example, denormalization would mean to copy the name of the business into the document of the deal against only referencing the business. The document of the business data embedded in the deal is called Place.

**4.2.4.1.2 Ratings and Comments** In the relational model an entity that is related to two or more entities is called a polymorphic association. DOM doesn't support polymorphic associations due to the fact that it would result in queries across collections. The solution to this is the aggregation method to embed documents in each other. In this application there exist the model of a *Comment* and a *Rating*, which both relates to a Deal or a Business and cannot exist without. In DOM this means that Comment and Rating should be embedded documents in a parent, which in this case is a Deal or Business.

**4.2.4.1.3 Members** The end users should also be able to register and login to the system as members in order to comment and rate the deals and businesses. To keep track of the member data, a *Member* collection is added as well.

These members should also be able to register as a owner of a business. This means that there need data for keeping track of the relation between members and businesses. The list of the businesses that the given member owns is listed in the member profile. This means that the business accounts is on the same level as the profile/information about the member and cannot exist without the given member, which is a indication that the BusinessAccount document should be embedded in the Member document.

**4.2.4.1.4 The Live Feed** The live feed is a feature that should be able to list the newest comments and ratings. In MongoDB it is not possible to query across collections, which means that this query is not possible with the data model at this point. A solution would be to query the Deal and Business collection separately and sort out the comments and ratings on application level. Although this should be a simple implementation it is not scalable and is highly correlated to the amount of deals and businesses in the databases. To solve this in a scalable way, the query need to happen on database level, which means that the comments and ratings should be unified in one collection. The normalized solution would be to aggregate the concept and extract the comments and ratings into a common collection. Although this could be a solution in another application, it would mean that the other features of the iPhone App still would need to query the LiveFeed collection. The solution is therefore to denormalize every Comment and Rating into their own common LiveFeed collection.

The denormalization means that there will be an update anomaly in the sense that every time a Comment or a Rating needs to be updated, the application would need to both update it in the parent document and in the LiveFeed collection.

**4.2.4.1.5 The Final Data Model** The final data model results in the following documents: Deal, Business, Comment, Rating, BusinessAccount, Place and LiveFeed. The relation between the different documents are illustrated in figure 4.4.

#### 4.2.4.2 Resource-Oriented Architecture

The REST architecture gives the interface of the web service a simple and transparent interface, such that the developers are able to understand the API without much documentation. In order for the web service to implement the REST architecture and thereby keep a RESTful interface, the API will be implemented by the constraints of the Resource-Oriented Architecture (ROA) that

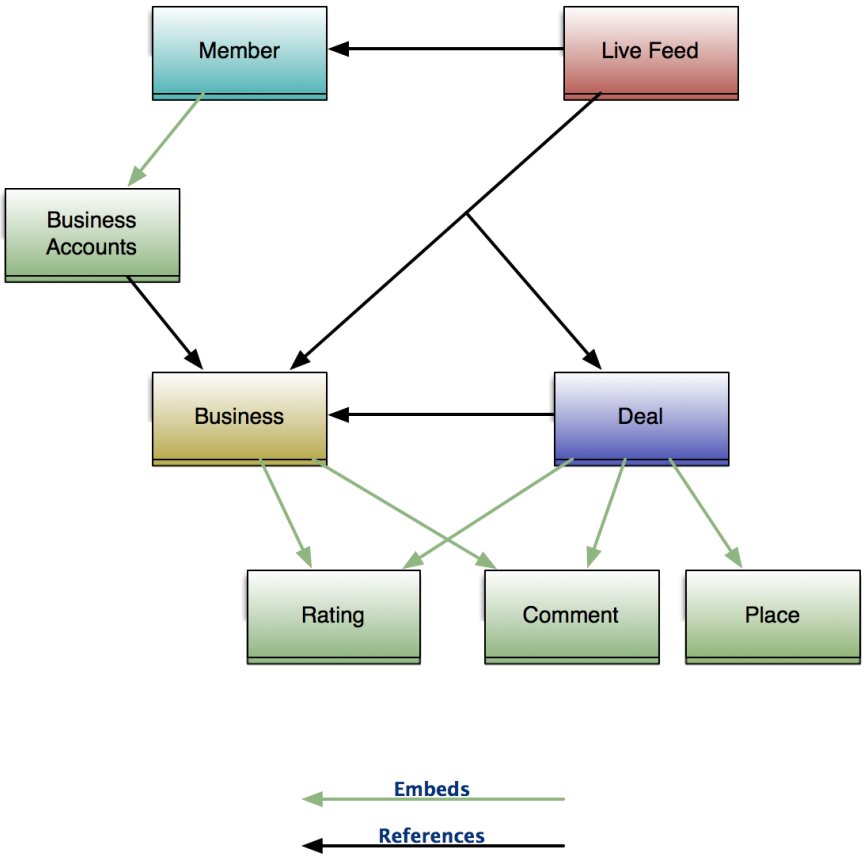



**Figure 4.4:** The data model of the application. The green arrows means that the document destination is embedded in the source and the black arrow means references.

was formulated by Leonard Richardson and Sam Ruby in *RESTful Web Services* [RR07].

ROA is a set of architectural constraints that helps developing a RESTful interface for the web service that has the properties *connectedness*, *addressability*, *uniform interface* and *statelessness* interface.

The connectedness property is not included in the architecture of this web service, because the API only is for internal use, which means that changes of the URI of a resource is easy to maintain without the connectedness.

In order for the developer and the user (if the target was a web application) to be able to derive the meaning of a URI, the URI of a resource needs to be descriptive - thereby addressable.

The uniform interface is to be able to derive the meaning of a resource and an HTTP-method. In ROA only the five HTTP-methods *GET*, *POST*, *PUT* and *DELETE* is used (*OPTION* and *HEAD* are also options that are not used in this project). The possible actions of a resource and the related HTTP-method is as seen on table 4.5.

The last property follows the REST-interface - statelessness. Statelessness means that there should not be saved any state of a session on a server. This could e.g. be done in order to save data about the user and his session. In a web application, there exists the concept of cookies, which is a way to save user state on the client of the user. In this web service no cookies are used. The only way, the server is able to retrieve state about the user is by authorization in the request. This property is a must have in a distributed system, where the load balancer should be able to direct the traffic to an arbitrary server.

| **Action** | **HTTP-method** |
|---|---|
| Retreive a representation of a resource | GET |
| Create a new resource | POST or PUT[4] |
| Modify an existing resource | PUT |
| Delete an existing resource | DELETE |

**Table 4.5:** Resource actions and the related HTTP-method in ROA

The resources of the web service of this project is as seen on table 4.6.

#### 4.2.4.3 Security

In the analysis in section 4.1.1.1 it was concluded that an attacker is able to read the request of users that are on an unsecure network. When designing for a fully robust system, the developer must be in a state of paranoia and not trusting the users, which means that an attacker probably will sniff the network traffic. In this section, the security design of the web service is discussed - at first with relation to securing the user's private information followed by the identity and the final design of the security system behind the web service.

| Resource | URI | HTTP-method |
|---|---|---|
| Create member (register) | /members | POST |
| Create a member session (login) | /members/sessions | POST |
| Create a deal | /deals | POST |
| Create a deal comment | /deals/:deal_id/comments | POST |
| Create a deal rating | /deals/:deal_id/ratings | POST |
| Get deals nearby | /deals/nearbysearch | GET |
| Get information about a deal | /deals/:id | GET |
| Create rating for place | /places/:place_id/ratings | POST |
| Get places by text query | /places/textsearch | GET |
| Get places nearby | /places/nearbysearch | GET |
| Places querysearch | /places/querysearch | GET |
| Register as business owner | /businesses | POST |
| Get information about a place | /places/:id | GET |
| Get information about a business | /businesses/:id | GET |

**Table 4.6:** List of resources and their URI of the web service. The URI-parts with a semi-colon means that it's a variable. E.g. to get information about the deal with id *abdcef*, the URI will be */deals/abcdef*.

**4.2.4.3.1 Protecting Private Information** When the user registers or needs to authorize, credentials needs to be transferred with the HTTP-request. In order to protect these credentials from the attacker that sniffs the network, they need to be blurred in such a way that the attacker cannot understand them. On the same time the credentials need to be understood by the web service. A solution to these requirements is to encrypt the credentials. A standard used for encrypting HTTP requests is *HTTPS*, which is HTTP with a *TLS/SSL* layer on top. HTTPS provide both full encryption of the whole request and additionally a certificate handshake to make sure that the sender of the response is correct. The TLS layer consists of several handshakes including hash encryption and decryption, which in theory should have an impact on the performance. Although this is true in theory, it should only be with a few milliseconds in practice as claimed by G. Coulouris, J. Dollimore, T. Kindberg and G. Blair [CDK05, p. 505].

One thing is to protect the private information in the network traffic, another

thing is to protect the information stored in the database and given in the resources. A hacker is able to retrieve the data from the database. If the password is saved as text in the database, the hacker would potentially be able to retrieve the password of every user, which is not acceptable due to the fact that many users use the same password for other logins. To prevent this, the password will not be stored in the database. The only thing that is stored in the database to authorize a given user is a checksum that can be used for verifying a given password. This way, the web service can still authorize a user by the given password, while the attacker cannot retrieve anything from the database.

**4.2.4.3.2 Protecting User Identity** With HTTPS as the solution for protecting the user's identity, it would be possible to transfer the email and password for every request, because they are encrypted anyway. But with this solution, HTTPS would be needed in every request.

If the HTTPS-protocol gives a performance penalty, another solution is needed. Instead of having to authorize with credentials every time, the client only need to authorize once with the credentials. At this authorization, the client will receive a session key, which can be used to authorize in further requests. The session model is illustrated at figure 4.5.

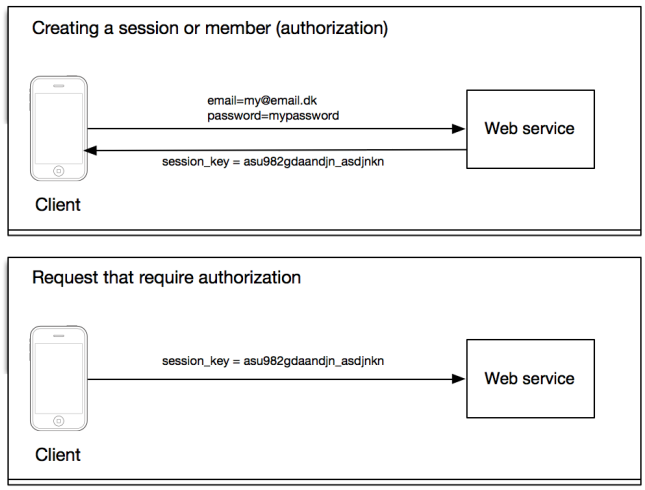



**Figure 4.5:** The session model, where the client authorizes with credentials and thereafter authorizes with the given session key.

Although the session model protects the user's private information, it still exposes the user's identity due to the fact that every client with a given user's

session key is able to send request on behalf of the given user. A solution to this could be to use the HTTPS protocol. Another solution would be to use time limited sessions, which means that the session key given on authorization only is valid in a given time interval. The time limited session model is illustrated on figure 4.6. This means that the session hijacker will be limited in the use of the session key. A downside to the time limited sessions is that every client has to design for HTTP-retries in order to renew the session keys, when the session key is out-dated.

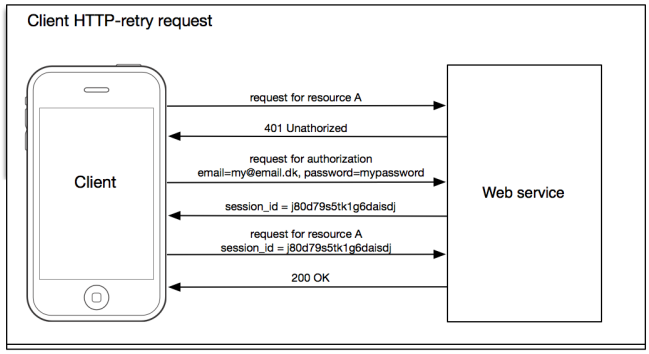



**Figure 4.6:** The time limited session model, where the client need to retry when unauthorized.

**4.2.4.3.3 Final Security Design** The comparison of the different authorization protocols for securing private information sent with requests is organized in table 4.7. It should be noticed that the different methods could be combined.

The simplest and most secure solution for securing HTTP request would be to use HTTPS on every request, which would mean that the attacker is not able to get information of any request. Although it is more secure, the HTTPS protocol have a rumour to perform badly and could cause caching problems. Another thing is that it requires a certificate from a SSL certificate provider that cost money.

The authorization protocol of the web service at the point of this project is therefore the simple session model as illustrated on figure 4.5 without time limited session keys. This means that if an attacker can sniff the traffic of a user, the attacker would be able to get registration info such as email and password on authorization and steal the identity on every other request by getting the session key. Although the authorization method of the web service is open for

| Method | Pros | Cons |
|---|---|---|
| Basic session model | • Simple to implement on both client and web service | • Target for session hijacking<br>• Email and password sent with plaintext on authorization, when without HTTPS |
| Time limited session model | • Leaked session keys can only be exploited in a short time period | • Complex to implement on clients<br>• Email and password sent with plaintext on authorization, when without HTTPS |
| HTTPS | • Implementation doesn't differ much from HTTP other than with deployment<br>• Encrypts all data in requests and thereby secures private information | • Performance penalty on certificate handshake<br>• Adds complexity to caching<br>• TLS certificates costs money |

**Table 4.7:** A list of pros and cons of the three authorization methods discussed in section 4.2.4.3

attackers on requests, the user's passwords won't be stored in the database as plaintext.

In future development, the HTTPS protocol will be profiled and implemented, if the performance penalty is not too bad. If the performance penalty weakens the user experience too much, the session model will be implemented with time limited session keys.

### 4.2.5 Database System

The design of the database system is next. The MongoDB architecture, the scaling strategies, the failover strategies and the recovery strategies are described in this chapter.

#### 4.2.5.1 Database Architecture

The natural architecture of MongoDB is master/slave as shown in figure 3.3. The master/slave setup is configured such that it is only one node is responsible for writes. This server is called the master. The other nodes in the system are called slaves and are all replications of the master. The slaves are only used for reads. This architecture is designed upon providing eventual consistency in the way that every record only has one master. In this project, the database is implemented as a native database, but in future development the database system will have its own nodes in a master/slave architecture.

#### 4.2.5.2 Scaling for Read Overload

When the database system needs to scale due to an overload of reads, the replication technique is used. When a new node is added to the system, it will sync up with the master and be assigned a slave role. The slave role means that every time the data of the master node changes, it will replicate itself to all the slaves in the system. This way all the slaves will eventually be consistent with the master node. The problems of modelling a database with eventual consistency is discussed further in section 4.1.4.

#### 4.2.5.3 Scaling for Write Overload

The solution of a CP database system as MongoDB that need to scale due to an overload of writes is the sharding technique. The sharding technique is described in section 3.6.3.2, but with regards to scaling, a sharded database will mean to have several masters, each responsible for their own data. This means that the masters doesn't share data and are thereby able to be partition tolerant and still be eventual consistency by maintaining a "1 record per master"-strategy. [SF12, chapter 9]

## 4.3 Implementation

At this point the system have been designed, which is followed by this section about the implementation of the system design. The implementation is large, so only the most important and advanced implementations are discussed. This includes features from the iPhone app, the authorization system of the web

service, the data integration with the Google Places API and the automated tests.

### 4.3.1 iPhone App

The most advanced and remarkable feature implementations for the iPhone app are described in this chapter.

#### 4.3.1.1 Advanced Map Search

Map search is a feature that gives the user the possibility to search on the map and find shops that is not nearby. Google Places takes a location coordinate and a radius. When finding nearby shop the GPS location of the user is sent and the 20 nearest and most valuable shops based on the user location is returned. This is not the case when navigating elsewhere on the map. Instead the center of the map is sent together with the calculated radius.

Calculation of the radius is necessary in order to give the best results that are visible on the map the user looks at. Therefore the results should differ if the user have zoomed out on Seeland compared to when the user have zoomed in on Nørrebro. The width of the specific map view, which is shown to user, is needed in order to find the right radius. Each degree of longitude is approximately 111300 meters. The width of the map view in meters is found by subtracting the longitude of the top right corner of the map view with the longitude of the button left corner of the map view and multiply it with 111300 meter. 2 is divided in order to get the radius.

The method for calculating the radius (see listing 4.1) returns the radius in meters and is later send to Goolge Places. A precision correction of a factor 0.60 is multiplied to the radius, because some shops was shown just outside of the map. This could be because the $111300 meters/longitude$ is only an approximation, and after numerous experimental testes 0.60 was the optimal factor.

Listing 4.1: Calculation of map view radius

```
- (NSNumber *)radiusInMeters
{
  CLLocationCoordinate2D topLeftCornerCoordinate,
       buttomRightCornerCoordinate;
  CGPoint topLeftPoint = CGPointMake(self.mapView.frame.size.width, 0);
```

```
    CGPoint bottomRightPoint = CGPointMake(0, self.mapView.frame.size.height)
        ;

    topLeftCornerCoordinate = [self.mapView convertPoint:topLeftPoint
                                  toCoordinateFromView:self.mapView];

    buttomRightCornerCoordinate = [self.mapView convertPoint:bottomRightPoint
                                       toCoordinateFromView:self.mapView];

    float distanceLonInDegrees = topLeftCornerCoordinate.longitude -
        buttomRightCornerCoordinate.longitude;

    float lonInMeters = 111300 * distanceLonInDegrees;

    float precisionCorrection = 0.60;
    float radiusInMeters = lonInMeters/2 * precisionCorrection;

    return [NSNumber numberWithFloat:radiusInMeters];
}
```

When the user updates with a new location the center of the map is the location coordinates and the half width in meters is the radius. As seen on figure 4.7 this results in that the shops that are outside of the radius still on the map, are not considered.

If the search were performed on the height of the map, the search would cover more of the map and thereby consider more shops that are visible to the user. But the problem with this is that if all the 20 most valuable shops where outside of the map no shops would be shown. The reason why it is troublesome to get perfect is that the Google Places only is able to consider map bounds by a radius and thereby a circle are, where the map view is rectangular. If OSM were chosen a more precise algorithm could be implemented.

#### 4.3.1.2 Ratings on Places and Deals

An end user is able to rate a place and a deal. This gives other end users an indication of the quality of the shop and the deal. The rating parameters of places and deals are the same in other words ratings are polymorphic. The rating feature works by tapping a button on either a deal view or a place view, where a rating interface will pop up and ask for the rating. When the user saves the rating, the controller of the rating view sends the given rating to the web service. This means that the rating view controller has responsibility for two kinds of ratings, which is not optimal in object-oriented programming. A solution could be to split the rating view and controller into two, such that there exists one for both deals and places. Thereby the responsibilities are separated, but it also

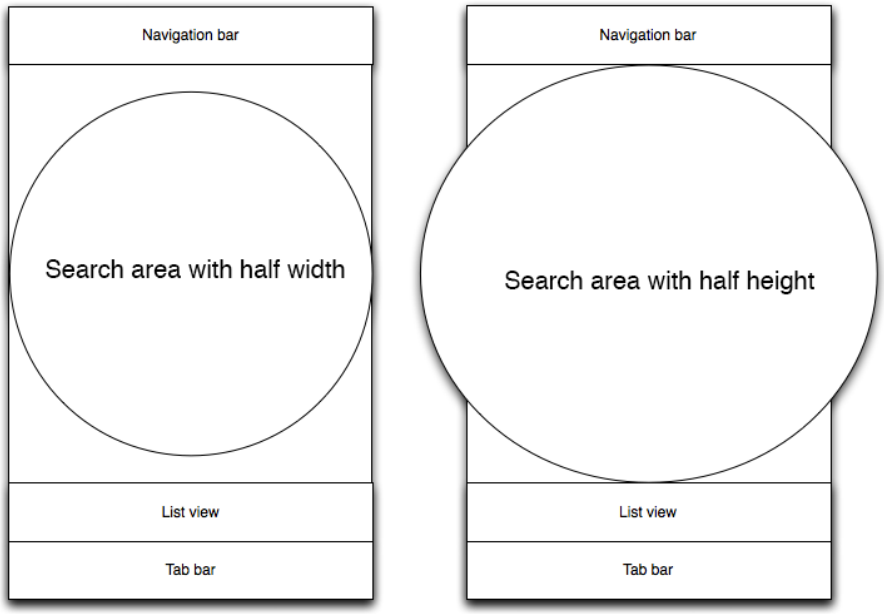



**Figure 4.7:** The search area problem is shown

means that if the interface of a rating changes, it should be implemented in both views and controllers, which is a bad programming practice.

The solution chosen for the problem was to have a rating view controller that only controls the view and does not send the rating requests to the web server. Instead the rating view controller notifies the responsible class (called a data source) with the relevant values. This data source could both be for a deal or a place, which means that for the rating view controller is a dynamic type. This could be implemented by the dynamic type in Objective-C `id` and the dynamic programming style *duck typing*, but this would require introspection in order to secure that the dynamic variable responds to the relevant methods. This is a problem that can be solved by the Objective-C programming pattern *protocol*. A protocol is a way of telling, which methods to expect for a given class. In this case the rating view controller has a data source property that should conform to a data source protocol. The data source is set by the parent view controller (here the detailed view of deal or place), when the rating view controller is allocated and shown. The protocol for the data source is shown on listing 4.2.

Listing 4.2: Saving the rating

```
@protocol MMRatingDataSource <NSObject>
```

```
@required
- (void)saveRatings:(MMRating *)rating
         completion:(void (^)(NSString *errorMessage))completionBlock;

@end
```

This protocol means that an object, which conforms to the `MMRatingDataSource`-protocol must have the `saveRatings:completion:` method implemented. This way, the rating view controller can execute this method on its data source safely, because the data source conforms to the data source protocol and thereby responds to the method. When the rating is submitted and the save button is pressed, the rating will be saved by the method shown on listing 4.3.

Listing 4.3: Saving the rating

```
- (IBAction)didPressSaveButton:(id)sender
{
  MMRating *rating = [[MMRating alloc] initWithServiceRating:self.
       serviceRating
                                          andAtmosphereRating:self.
                                              atmosphereRating
                                              andProductsRating:self.
                                              productsRating];

  [SVProgressHUD showWithMaskType:SVProgressHUDMaskTypeClear];

  MMRateViewController *weakSelf = self;
  [self.dataSource saveRatings:rating completion:^(NSString *errorMessage)
       {
    if (errorMessage) {
      [SVProgressHUD showErrorWithStatus:errorMessage];
    } else {
      [weakSelf.delegate didUpdateRating];
      [SVProgressHUD showSuccessWithStatus:@"Thank you for your opinion!"];
      [weakSelf dismissViewControllerAnimated:YES completion:nil];
    }
  }];
}
```

A big advantage of this is that the compiler knows about the protocol and warns the developer, if a given class does not implement the methods its protocols require.

When the rating is saved, the parent view controller should be notified that a new rating has been submitted in order to update the rating in the UI. This is implemented by another protocol called `MMRatingDelegate` that requires the `- (void)didUpdateRating` to be implemented. The execution of the notification

happens if the request succeeds as seen on listing 4.3.

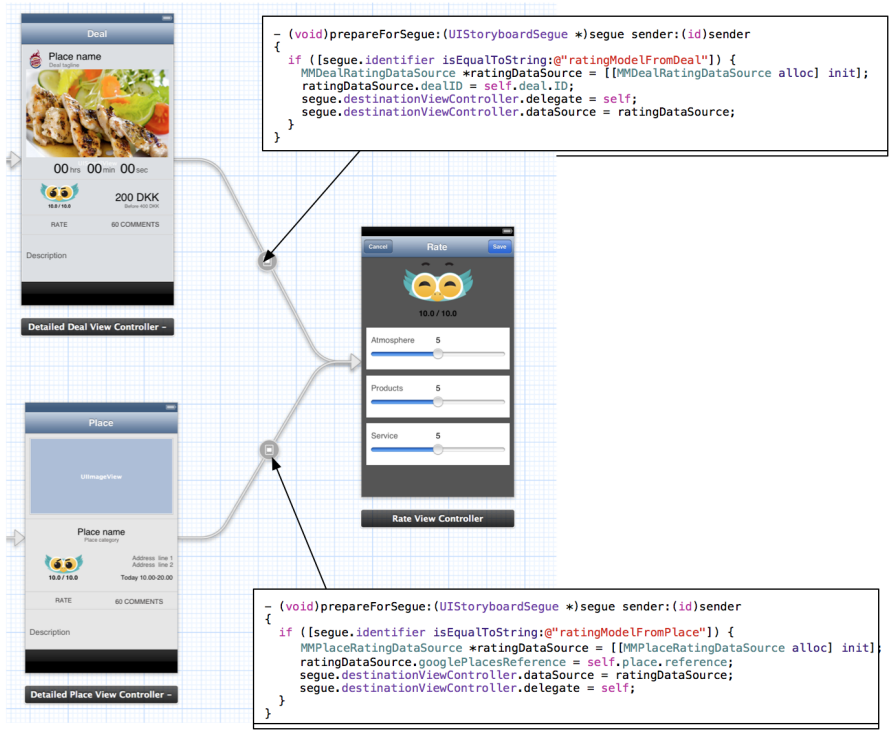



**Figure 4.8:** The solution for the polymorphic rating view that is used by both the rating and deal view controller. The rating view controller is activated by a modal segue, where the delegate and data source are set when preparing for the segue.

The implementation is illustrated at figure 4.8 that is an edited screenshot of the storyboard, where the rating view controller is a modal segue and activation of the modal are distinguished by how the delegate and data source are set in the *UIViewController* method `prepareForSegue:sender:`.

The Objective-C protocol pattern can be used for communication between classes that only knows the fact about each other that they conform to the given protocol and thereby the methods of the protocol. In this case it is used for separating the concerns in a polymorphic view controller.

### 4.3.2 Web Service

The web service was implemented with Ruby on Rails. The four main parts of the web service implementation was data modelling, the authorization system, deployment and the Google places integration. The data modelling was implemented with the Ruby library Mongoid and with no major issues. The three other parts will be explained in the following sections.

#### 4.3.2.1 Authorization

The authorization is based on the simple session model explained in section 4.2.4.3. For the user to access resources that requires authorization (such as creating a user comment or creating a rating), the user must at first be granted a session key. This implementation can be divided into two parts:

1. Authorization of user with credentials (request for session key)
2. Authorize the the user with session key

**4.3.2.1.1 Authorization with Credentials** The first step in authorization is for the client to request for a session key. The session key is an attribute of a session, which means that in order to get a session key for a given user, a session referencing to the given user must be created. In this system one can only authorize as a member, and the session model is therefore called MemberSession as explained in section 4.2.4.1.

The simple session model could be implemented by using custom encoders and decoders, but HTTP and Rails already supports a authorization schema called Basic Authentication and Token authentication, which is used for authorization in the system.

The authentication of credentials is implemented through the Basic Authentication Scheme[For99]. Basic authentication uses the standard *Authorization* header from HTTP. The content of the header is "Basic" followed by a space and the credentials. The credentials is a simple encoding of the username (in this system: email) and password separated by a colon.

A session is created both in its dedicated resource and when a member is created (registration). If a user wans to authorize with the email "my@email.com" and password "mypassword", the credentials will be encoded as follows:

```
email=my@email.com
password=mypassword
Base64('my@email.com:mypassword')=bXlAZW1haWwuY29tOm15cGFzc3dvcmQ=
```

The resource for authorization is `/members/sessions` as described in table 4.6. The request for authorization as the above example user is as follows:

**Example 4.1** *(An example of an authorization request)*

```
POST /members/sessions HTTP/1.1
HOST: v1.limecode.dk
Authorization: Basic bXlAZW1haWwuY29tOm15cGFzc3dvcmQ=
```

If the above user wanted to register, the request would be as follows:

**Example 4.2** *(An example of a registration request)*

```
POST /members HTTP/1.1
HOST: v1.limecode.dk
Authorization: Basic bXlAZW1haWwuY29tOm15cGFzc3dvcmQ=
```

The web service implements the authorization through basic authentication by the class `ActionController::HttpAuthentication::Basic` from the Rails API. The authorization is implemented as a method that is executed in every controller action that requires authorization of a member. The method for authorizing with credentials is as follows:

```
def authentication_with_credentials
  authenticate_with_http_basic do |email, password|
   MemberAuthentication.authenticated?(email, password)
  end
end
```

With this method, Rails decodes the encoded credentials. `MemberAuthentication\#authenticat` returns true and authenticates a client, if the credentials are correct. If the credentials aren't correct, there will be thrown a `MemberAuthenticationSystem::Unauthorized` exception, which is catched by the controller and converted into an error message for the client with the response code 401.

**4.3.2.1.2 Authorizing with Session Token** After authorization as described above in section 4.3.2.1.1, the client will receive a session key that can be used for authorization. Rails has expanded the support for the HTTP Authorization header with support for tokens, where the value to be authorized is a token. An example of a resource that requires authorization is for creating deal comments. If the client receives the session key *asdsaoixu89762tgkhjsg87ax*, the request will be as follows:

**Example 4.3** *(An example of an authorization request)*

```
session_key=asdsaoixu89762tgkhjsg87ax

POST /deals/comments HTTP/1.1
HOST: v1.limecode.dk
Authorization: Token token="asdsaoixu89762tgkhjsg87ax"
```

The implementation of token authentication is implemented by the Rails class `ActionController::HttpAuthentication::Token` . The implementation is almost as with basic authentication, but instead of email and password, the value is a token. The method that is executed as the before method is as follows:

```
def authentication_with_token
  authenticate_with_http_token do |token, options|
    MemberSessionAuthentication.authenticated?(token)
  end
end
```

As with authorization of credentials, this method does nothing if the token is valid and throws an exception if not.

### 4.3.2.2 Deployment

In order for the real clients to be able to communicate to the web service in production, the web service needs to be deployed onto a cloud server. In the development fase, the cloud provider was chosen to be Cloud.dk. Cloud.dk just expose a server with an IP address, but in order to use the server for a Rails web service, a the server need to be configured to handle HTTP on the web.

The web server setup for this web service was chosen to be nginx on top of unicorn. Advanced setups for nginx and unicorn for optimizing the system with

regarding to connections and deployment has not been a focus in this project and is therefore not further discussed in the thesis. The deployment procedure is implemented with focus on simplicity and works as follows:

1. Download the latest version of the source code from the master branch on GitHub
2. Symlink the downloaded version to be the "current" version of the application on the server
3. Restart all processes with unicorn

#### 4.3.2.3 Google Places Integration

In order to have data from the launch of the system, the Google Places API is used. Although the main data is taken from the API, the system requires being able to merge data from the system with the data from the Google Places. The data that should be merged between the system and the API are the data about the businesses and the business categories.

**4.3.2.3.1 Merging the Business Data** Having two sets of data is not an optimal solution. If both data set should be queried, there need to be queries on both data sets followed by a merge of the results. This is complex regarding implementation and performance. Another problem is consistency. If there exists shared attributes between the data sets, it is hard to know, which data set has the newest data. The optimal solution is to merge the two data sets into one, but fetching the data from Google Places and merging it into the systems database would be data theft and is therefore not a valid solution. Another solution is to submit the data from the system into Google Places, but the data submitted to Google Places goes through a long time validation process, which means that the new data will not be displayed before it is validated.

The solution ended up with being to merge the data on runtime as illustrated on figure 4.9. At first the search query is send to the Google Places API, which will return the relevant data about the businesses from Google. The business data in the database has a reference to the corresponding data in Google Places such that the business data corresponding to the data from the API search can be merged.

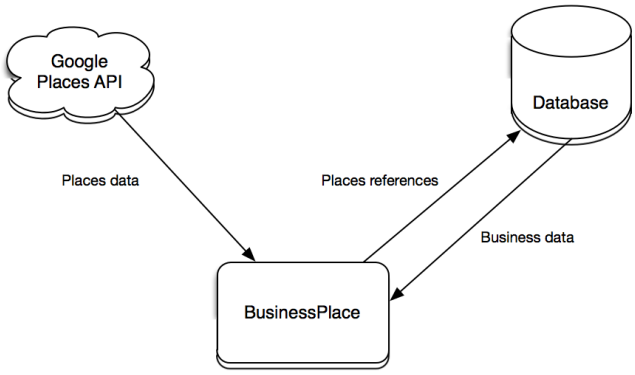



**Figure 4.9:** Illustration of how the data from the Google Places API and the database is merged

**4.3.2.3.2 Merging the Categories** The businesses in the Google Places API have an attribute called *types* that lists the categories of the business. In the system of this project, there should be custom categories. The custom categories are listed in the UI of the client. This means that a user e.g. is able to request for the businesses by the custom category "Nightlife". The Google Places search query will include the types related to the custom category as defined in the web service. For example the search for the custom category "Nightlife" will only consider the types "bar", "*night_club*" in the API search query, because the custom category "Nightlife" consists of the types "bar" and "*night_club*".

### 4.3.3 Automated Testing

In the design section 4.2.1.2 the test methods considered in this project were prioritized with relation to their importance and thereby when they are used. This section will describe the implementation of unit testing, UI-testing and controller testing in the web service.

#### 4.3.3.1 Unit Testing

A unit test is an automated piece of code that calls a method and then checks a single assumption about the behaviour of that method. There exist approaches to software development that focuses on letting the code be driven by development. One variant is test-driven development (TDD) where the developer

first writes a failing automated unit test with the desired properties. Then the minimum amount of code to pass this test is written. A variant of TDD called behaviour-driven development (BDD) is used in this project. BDD focuses on the behavioural aspect of the system rather than the implementation aspect of the system that TDD focuses on. BDD gives a clearer understanding as to what the system should do from the perspective of the developer and the user. TDD only gives the developer an understanding of what the system should do. This chapter introduces RSpec (for Ruby) and Kiwi (for Objective-C), which are testing tool under BDD that have been used in this project.

#### 4.3.3.1.1 RSpec in Ruby

RSpec is a BDD testing tool for Ruby, which is used in the web service, because of the high readability and automatic documentation. [webd]. An example of an RSpec unit test

Listing 4.4: Example of an Ruby Unit test with RSpec

```ruby
describe Member do
  describe "validations" do
    it "validates a Member given valid attributes" do
      member = FactoryGirl.build(:member)
    end

    it { should be_timestamped_document }

    it { should embed_many(:business_accounts) }

    [:email, :password_digest].each do |attribute|
      it { should validate_presence_of(attribute) }
    end

    it "invalidates a Member given invalid email" do
      member = FactoryGirl.build(:member, email: "myemail.com")
      member.should_not be_valid
    end

    it "invalidates a member given a email address already used" do
      FactoryGirl.create(:member, email: "my@email.dk")
      copycat_member = FactoryGirl.build(:member, email: "my@email.dk")
      copycat_member.should_not be_valid
    end
  end
end
```

This test validates registered members with email and password. The test invalidates a member with both an invalid email and an email address that has already been used. A developer without Ruby on Rails experience would be able to read the automated test, because of the natural English language.

**4.3.3.1.2 Kiwi in Objective-C** Kiwi is a BDD library for iOS development and tests are written like RSpec. This way the tests are more readable compared to the native test environment within Xcode. The developer experiences a consistent test environment, because the tests in the web service and on the application platform are written in the same way.

Listing 4.5: Example of an Objective-C Unit test with Kiwi

```
describe(@"Time counter", ^{
    it(@"is able to instantiate with a start time", ^{
        MMTimeCounter *timeCounter = [[MMTimeCounter alloc]
        initWithTimeIntervalInSeconds:60.0];
        [timeCounter shouldNotBeNil];
    });

    it(@"saves the start time on initialization", ^{
        MMTimeCounter *timeCounter = [[MMTimeCounter alloc]
        initWithTimeIntervalInSeconds:60.0];
        [[theValue(timeCounter.timeIntervalInSeconds) should] equal:
             theValue(60.0)];
    });

    it(@"decreases with one second", ^{
        MMTimeCounter *timeCounter = [[MMTimeCounter alloc]
        initWithTimeIntervalInSeconds:60.0];
        [timeCounter decreaseOneSecond];
        [[theValue(timeCounter.timeIntervalInSeconds) should] equal:
             theValue(59.0)];
    });

    describe(@"converting the time to string", ^{
        it(@"returns 59 seconds if the time is 59 seconds", ^{
            MMTimeCounter *timeCounter = [[MMTimeCounter alloc]
            initWithTimeIntervalInSeconds:59.0];
            [[[timeCounter toString] should] equal:@"00:00:59"];
        });

        it(@"returns 1 minute if the time is 60 seconds", ^{
            MMTimeCounter *timeCounter = [[MMTimeCounter alloc]
            initWithTimeIntervalInSeconds:60.0];
            [[[timeCounter toString] should] equal:@"00:01:00"];
        });

        it(@"returns 1 hour if the time is 3600 seconds", ^{
            MMTimeCounter *timeCounter = [[MMTimeCounter alloc]
            initWithTimeIntervalInSeconds:3600.0];
            [[[timeCounter toString] should] equal:@"01:00:00"];
        });
    });
});
```

This test (or spec) tests the time counter and secures that the time of the count down is correctly formatted.

#### 4.3.3.2 UI Testing

The iPhone app user interface can be automated tested. It enables the developer to track regression and performance issues and the developer is able to develop new features without worrying if it breaks the app. This is done in instruments by writing script in JaveScript. These scripts simulate user actions by calling UI Automation, and specify the actions to be performed in the app as it runs. During the tests, the system returns log information.

Here is an example of how to UI tests that the label changes from "Login" to "Profile" when a user logs in.

Listing 4.6: An example of UI test with JavaScript

```
var target = UIATarget.localTarget();
target.frontMostApp().navigationBar().buttons()["Login"].tap();
target.frontMostApp().mainWindow().tableViews()[0].cells()[0].textFields()
    [0].tap();
target.frontMostApp().keyboard().typeString("hej@hej.com");
target.frontMostApp().mainWindow().tableViews()[0].cells()[1].
    secureTextFields()[0].tap();
target.frontMostApp().keyboard().typeString("hej");
target.frontMostApp().mainWindow().tableViews()[0].cells()["Login"].tap();

var labelName = target.staticTexts()[0];

if (labelName == "Profile"){
    UIALogger.logPass("UITest4 - Login label did change.");
}else{
    UIALogger.logFail("UITest4 - Login label did not change.");
}
```

More comprehensive UI tests than this have not been implemented for the iPhone app. A whole test environment had to be sat up with fixed data. The reason for this is that the app communicates and dependents on the web service. The view changes rapidly which means that new tests have to be written. That is the reason why UI testing has not been a focus in the iPhone app development. Tests should not be written because of tests, but because of higher maintenance and code quality.

#### 4.3.3.3 Web Service Controller Testing

With Unit-tests the internal components of the system are tested as isolated units. For testing the actual application interface of the web service, the resources must be tested. When a resource is requested, it is the related controller in the Rails application that is responsible for handling the request input and response. What happens internally in the system is not the responsibility of the controllers and is therefore not tested. In order to separate these concerns, the controllers contain minimal application logic.

The things that are tested in the controllers are:

- Commands executed
- Parameters in a request that cannot be accessed for the given client are ignored.
- Correct status codes in responses
- Error messages in the response
- Exceptions are handled correctly

An example of a controller test in the application is seen on listing 4.7. This example is of the controller for creating sessions (authorizing). This spec tests for the cases where correct, wrong and no credentials are given.

Listing 4.7: Example of an Rails controller test in RSpec

```
require 'spec_helper'

describe V1::Members::SessionsController do
  let(:member) { FactoryGirl.create(:member, password: "secret") }

  before :each do
    request.host = "v1.test.host"
    request.accept = "application/json"
  end

  describe "POST create" do
    context "member is authenticated" do
      before :each do
        authorize_as_member(member, "secret")
      end

      it "creates a member session" do
        SessionService.
```

```
            should_receive(:create_for_member).
            with(member, request)
          post :create
        end

        it "responds with response code 201" do
          post :create
          response.code.should == "201"
        end
      end

      context "member is not authenticated" do
        before :each do
          authenticate_with_basic_auth("my@fakeemail.dk", "mywrongpassword")
        end

        it "responds with an error message" do
          post :create
          response_body_as_json.should include_keys ["errors"]
        end

        it "reponds with response code 401" do
          post :create
          response.code.should == "401"
        end
      end

      context "no authorization given" do
        it "responds with an error message" do
          post :create
          response_body_as_json.should include_keys ["errors"]
        end

        it "reponds with response code 401" do
          post :create
          response.code.should == "401"
        end
      end
    end
  end
```

## 4.4 Results

In order to verify some of the requirements, the system have been user tested and battery tested for user experience and scalability by load testing.

### 4.4.1 User Tests

User testing is a method of testing and improving the design and overall user experience. This is done through careful study of real live user sessions. Bug finding, detect areas of confusion and getting a fresh set of eyes on the applications are some of the many benefits with user testing. The iPhone app has had a short qualitative feedback from four different kinds of users. An independent interviewer did one of the interviews and these sessions have been recorded. The best examples of the video sessions can be downloaded by the links bellow:

1. Nikolai Jensen
2. Mads 3. video
3. Lars Sjælland 1. video
4. Andreas Graulund 1. video

The rest of the videos can be downloaded by the links in appendix D.

The setup to this test was simple. The test user had never seen the app before and was told to do certain actions without support.

The results of these user test indicated a clear pattern:

- 3/4 testers had problems creating deals with the right attributes on first try because of lack of description.
- 2/4 testers discovered and mentioned more than one bug or design confusion.
- 3/4 testers had an *wow effect* because of the design and user experience.
- 4/4 testers could see value in the iPhone app and would use it daily.

### 4.4.2 Scalability Tests

Requirement 5 under non-functional requirements in section 2.2 states that the system should be able to handle a maximum of $10,000$ in a time-interval of 1 min. It was discovered early in the project that the Cloud.dk deployment was not able the scale this amount of users and scalable deployment was not in the

scope of this project. The codebase was therefore deployed to Heroku in order to show how simple scaling with an IaaS is.

For testing these requirements, the Apache HTTP server benchmark tool version 2.2 was used. [webe]. The tests were performed with 10,000 requests with 50 concurrent connections at a time. This is performed with Apanche benchmarking tool with the following terminal command:

```
ab -n 10000 -c 50 -r URL
```

The tests was performed with a MacBook Pro with a 2,4 GHz Intel Core i7 processor and 8 GB 1600 MHz DDR3 RAM and compared between the two providers: Cloud.dk and Heroku. The requests were only for testing the connection to the web service and was therefore simple with no database interaction.

On Cloud.dk the tests were performed on a Virtual Private Server (VPS) with a Micro server of 512 MB RAM and 1 CPU. [weba] The OS of the VPS was Ubuntu 10.04.

On Heroku test tests were performed on up to 6 dynos, where each dyno is of 512 MB RAM and 1 CPU. [webf].

The results is presented on table 4.8 and compared in column diagrams comparing requests per second on figure 4.10 and the average response time on figure 4.11.

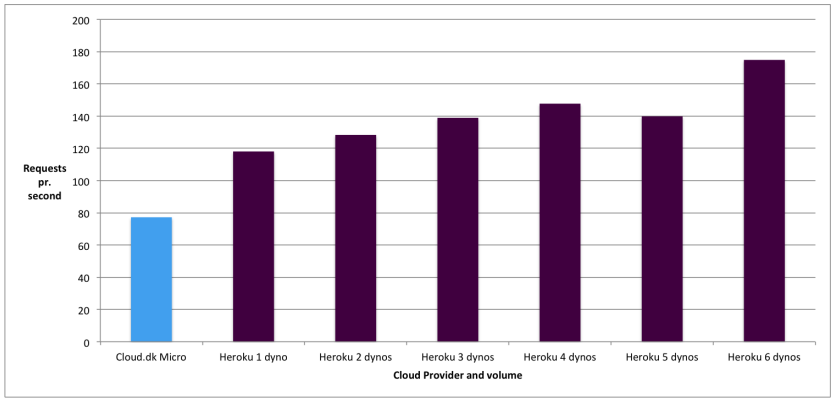



**Figure 4.10:** The amount of requests per second of the tests compared between different cloud provider and volumes

| Provider | Time (s) | Req/sec | Avg. response time (ms) |
|---|---|---|---|
| Cloud.dk Micro | 77 | 130 | 658 |
| Heroku 1 dyno | 118 | 85 | 357 |
| Heroku 2 dynos | 128 | 78 | 390 |
| Heroku 3 dynos | 139 | 72 | 370 |
| Heroku 4 dynos | 148 | 68 | 339 |
| Heroku 5 dynos | 139 | 72 | 358 |
| Heroku 6 dynos | 175 | 57 | 286 |

**Table 4.8:** Cloud.dk and Heroku compared with different volume based on time, requests per second and the average response time.

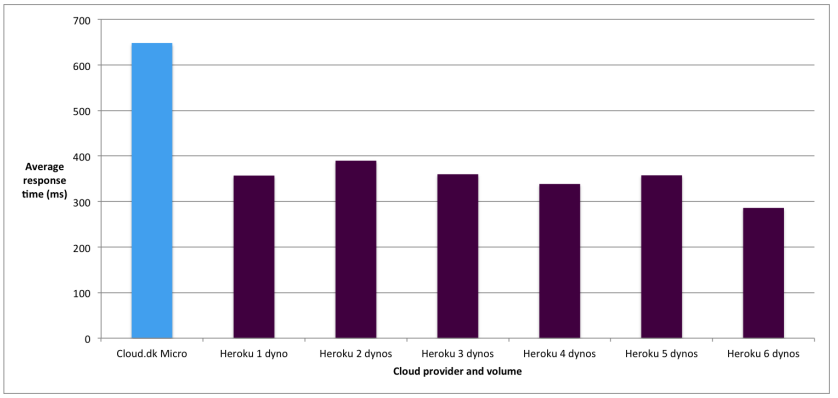



**Figure 4.11:** The average response time of the tests compared between different cloud provider and volumes

### 4.4.3 Battery Tests

When testing the battery usage on an iPhone app many source of error can occur. By measuring the power usage in Instruments in Xcode a precise result of the power usage can be measured, but that is not a real world use case. Instead an open source power usage measurement app called Powergremlin[5] were used.

The app was use in 10 minutes and Powergremlin measured the precise bat-

[5] `https://github.com/palominolabs/powergremlin`

| App | Time of Use (min) | Used Battery (mAh) |
|---|---|---|
| YouTube | 10.00 | 65 |
| MyModo | 10.15 | 66 |
| Facebook | 9.96 | 121 |

**Table 4.9:** Comparison of battery usage between the App of the Project (MyModo), YouTube's and Facebook's

tery usage. As comparison the YouTube app and the Facebook app was also measured. The iPhone was restarted after each session and no other apps than the test target and Powergremlin was running. All the tests were ran with the same WiFi connection and same level of screen brightness. There was not the same level of battery capacity during the different tests. If these had an effect is doubtful, but unknown.

As seen in the result table 4.9, YouTube and our app (MyModo) use approximately the same amount of battery capacity (65 mAh and 66 mAh), where Facebook uses twice as much.

## 4.5 Discussion

Although the user test was limited due to too few users there are clear indications that the user experience gave a unique experience with few glitches. Still more development and designing hours are needed before launching to App Store, but that the overall app experience gave unique value. More sufficient proof should be collected before the app is launched to the App Store.

From the comparison of the amount of requests processed per second on figure 4.10 it can be observed that it is easy to scale the web server horizontally with Heroku. The comparison of Cloud.dk and Heroku as cloud provider is not fair, because Cloud.dk was not scaled horizontally. This test was more a prove of how easy it is to scale with an IaaS as Heroku compared to setting up the servers ourselves. Table 4.8 shows that with the test of 50 concurrent requests it was possible to finish all 10000 requests in under 1 minute with a response time of under 500 ms and thereby meet requirement 5 and 10 (for that simple request). But as shown on figure 4.11, the response times did not differ much between the different amount of dynos on Heroku. One reason cloud be because the servers was not stressed enough (there was not send enough concurrent re-

quests) such that there was not many connections waiting the request queue or because Apache benchmarking tool became a bottleneck. This means that the requirements probably could have been fulfilled with less dynos. The number of dynos needed in order to meet the requirements could also have been optimized by e.g. trying the application with JRuby on JVM or test if vertical scaling (supported in beta on Heroku at the moment [webg]) to see if it is cheaper.

Concerning use of battery our test indicates that compared to other apps we are not off limits. Even though the battery usage test is not a complete and satisfactory test. The tester tried to use the different apps as much as possible in the same way, but it was only for 10 min usage and with one iPhone. If this should have been a complete real world test, multiple iPhones and more time with the apps should have been used. Still this test gives a hint that our iPhone app does not use too much battery. The developer should have focus on the battery use when implementing by e.g. turning off the location manager that handles the GPS when not needed and overall not implementing unintelligent solutions that drains the battery. This is the best way to avoid abnormal battery usage of an iPhone app.

The system robustness have been analyzed and designed with a goal to have a scalable, highly available, easy maintainable system. In order to have all these properties for a system every component need to have them. As explained in the analysis in section 4.1.1.2, the image delivery component is highly available by the commitment of the CDN provider. The current setup with a web service and native database system on Cloud.dk is neither scalable or highly available due to the many single points of failure such as the native database implementation and a cluster of only a single node. But even if the current deployment is not scalable, we have acheived knowledge of how to get a highly available and by deploying the system to services such as PaaS and DaaS it would be possible to have a easily scalable and highly available system due to the horizontal scaling. The maintainability of the system is acheived by choosing a well fit software architecture, reviewing each others code and setting up an automated testing environment.

## 4.6 Conclusion

Every functional requirement have been met through out the features of the iPhone app. The system was also designed to meet the non-functional in section 2.2. Table 4.10 shows an overview of the different non-functional requirements and how they were met.

| Req. no. | Met? | How |
|---|---|---|
| 1 | X | Ruby on Rails was used for the web service |
| 2 | X | HTTP was used as the communication protocol between client and web service |
| 3 | X | Tests in section 4.4.1 indicate that the iPhone app did not use abnormal battery |
| 4 | (X) | Images will be stored in the CDN and data in MongoDB. The CDN is outsourced and meets the requirement. MongoDB should be able to handle 10000, but needs to be verified with tests. |
| 5 | X | Tests in section 4.4.2 proves that the web service could handle simple requests from 10000 end users in a time scope of 1 minute with the use of Heroku. |
| 6 | X | Tests in section 4.4.2 proves that the web service could scale horizontally with the use of Heroku. |
| 7 | (X) | With use of MonogoHQ the database system can be scaled horizontally, but MongoHQ have not been used. |
| 8 | | Have not been met because the CDN have not been implementated yet due to the cost |
| 9 | (X) | The image handling is outsourced to a CDN, so it is just a matter of cost. |
| 10 | X | Tests in section 4.4.2 proves that a simple request maximum takes 500 miliseconds with use of Heroku. The tests was without interaction with the use of the database. |
| 11 | X | As described in section 4.2.4.3 it is not possible to retrieve a member's password from the database. |
| 12 | X | As described in section 4.3.2.1 the session model was used. |
| 13 | X | As described in section 4.3.3 automated tests have been implemented. |

**Table 4.10:** The non-functional requirements from section 2.2 and how they were met.

The reason for the robustness of the system is that the user should get a good and secure user experience, but also so the system is maintainable.

The different features in the iPhone app were chosen in order to give different kind of value to the end user and the business user. Only user experience techniques that had been statistically proven were chosen in order to get happy users.

Location data was chosen because an app that shows shops without shops is not a useful app. The Google Places API was chosen because of the easy access, but also because more data are provided compared to OSM. Though problems with Google Places occoured both in the design but also in the implementation which perhaps gives reason to change later.

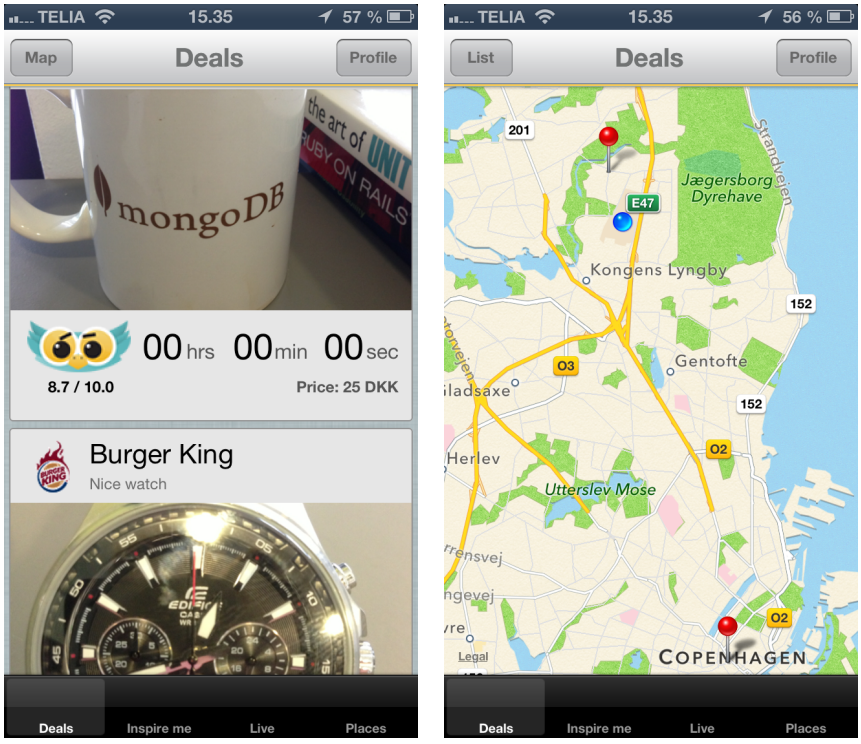
Chapter 5

# Final Product

We are excited to show the final product, this gives an idea of what the product is about. The final product is presented by us in the following video (the same as given in the summary):

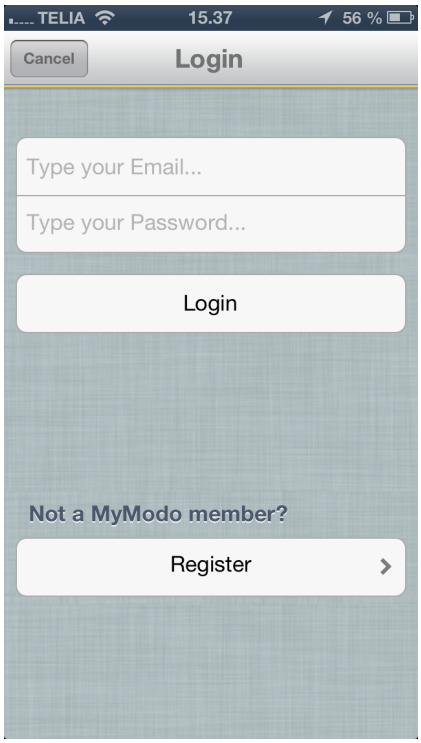
https://www.dropbox.com/s/ons2botut31bnf1/mymodo_app_presentation.mov

The following pages are screenshots of the different views.

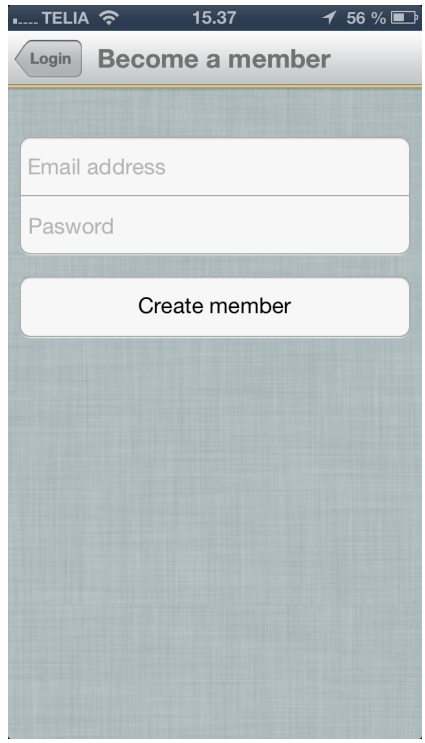



**(a)** List view of nearby deals **(b)** Map view of nearby deals

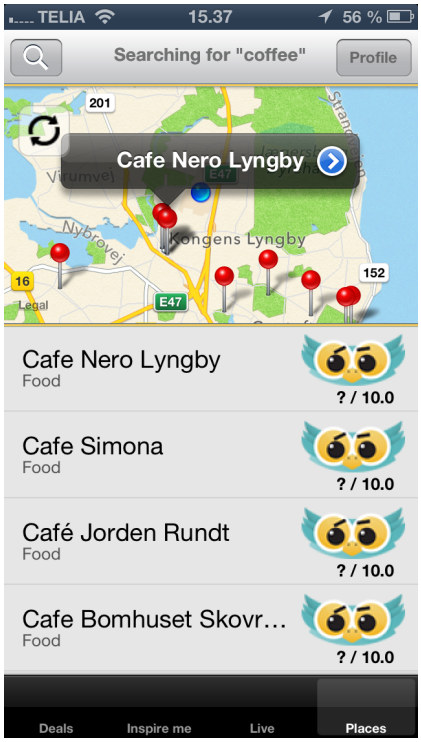
**Figure 5.1:** This shows how deals are displayed




**(a)** Member login

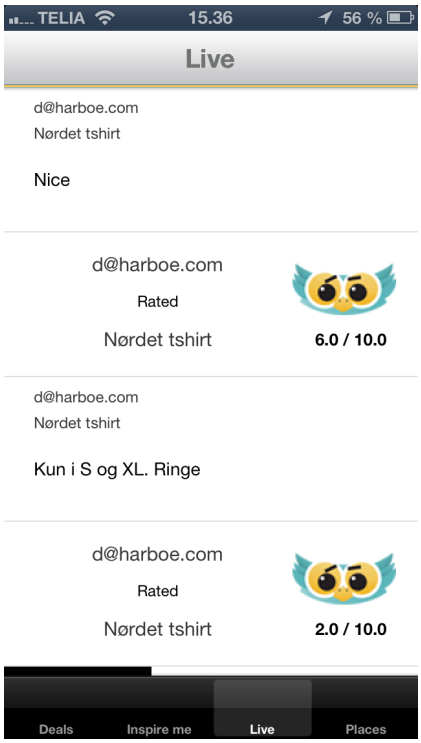



**(b)** Member registration



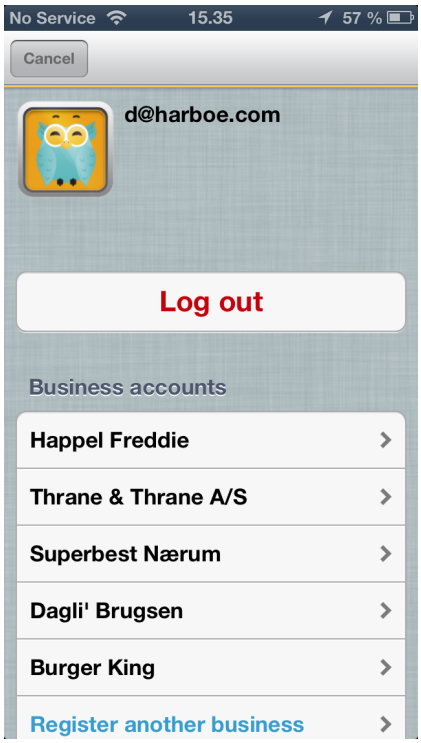

**(a)** Places with both map and table view when searching for "coffee"


**(b)** Searching for "pizza" other places on map




**(a)** The Live feed (not designed yet)


**(b)** Inspire Me feature




(a) The profile


(b) Business registration

Chapter 6

# Process

The project was developed as a collaborative project, which required that the team (of three people besides us) agreed on the overall product development decisions. Besides working in a team of non-developers, there were also challenges related to developing the system in a team of two persons.

Overall the project was divided into two parts. The first part was expanding the idea into mockups in collaboration with the team. The second part was to develop the system based on the mockups, which

The first part is described more in section 6.1. In the second part of the project it was planned to use the agile development method scrum. But since the time of the other team members were limited, most of the development were driven by us, so the tools from scrum was not used for the development. The internal development of the product was more inspired by the Kanban method, where the team works on the most important features of a prioritized list instead of timeboxing the features as with Scrum. The primary tool for Kanban is a board with features from the backlog divided into which state of the development process they are in.

The following sections will go through the process of how we in collaboration with the team came from a simple idea to an iPhone app that was ready for App Store followed by how we as developers managed the feature backlog and

implemented the iPhone app in collaboration.

## 6.1 iPhone Development

The main idea and the purpose of the app is settled. The next phase is how to make it a reality. This chapter goes through the different step of the iPhone app process without implementation. The main reason for this process is to save time on implementation, which is the most time consuming. The clearer overview of the app, the easier and the less time is needed on implementation. Obvious misunderstandings and implementation mistakes can be avoided when having a paper drawing of the UI flow and a prioritized feature list.

### 6.1.1 From Idea to Features

Every software application starts with an idea. In this case the application platform and because of the concepts' originality, it was not possible to get direct inspiration from similar apps. The process of going from an abstract idea to deciding the most important features is not simple. Everyone had different ideas of how the iPhone app should be designed. First of all the team had to agree on the value proposition. What value does this product bring? When the value proposition where decided, then which necessary features in order to meet the value proposition should be in version 1.0. Everyone had a stack of Post-its and every feature was welcome. All the features were sat on to the wall in an "Icebox". These Icebox features should then be prioritized in a order of what is most important as seen in the photo 6.1. After this session, which took two days, the most important and necessary features for version 1.0 where settled.

### 6.1.2 From Features to Mockup

With user experience in mind, as described in the analysis 4.1.3, the user interface flow were next. Decisions on what happens when that button is pressed and what should the user see as the first were made. The features had to be prioritized even further in order of what were most important features of all the features. The most important feature should give the biggest value to the user and therefore should be the first view when the app is opened. With simple tools such as paper and tape a simple flow view was drawn.

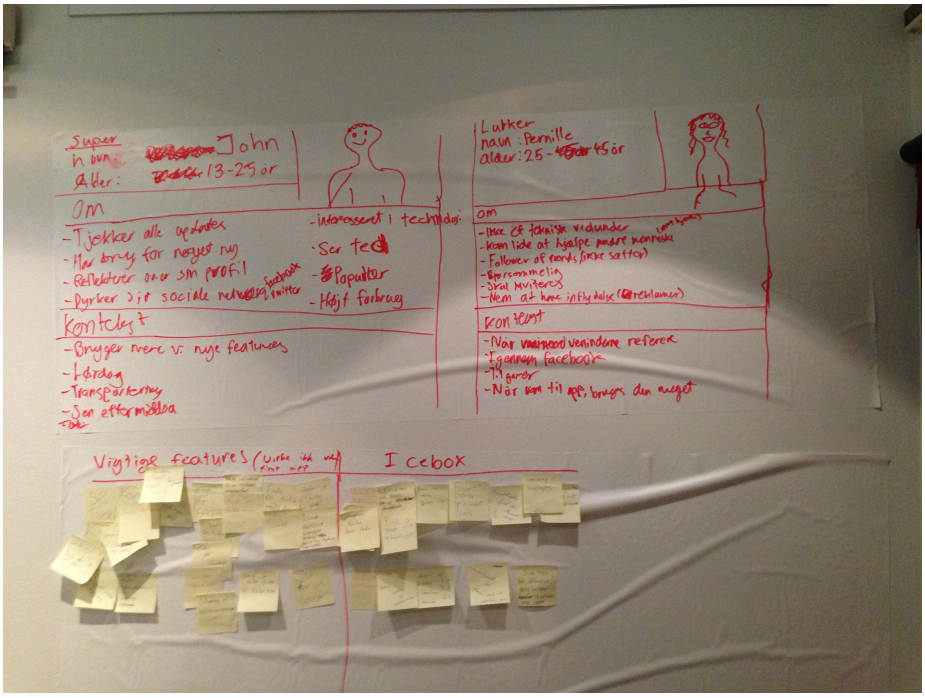



**Figure 6.1:** Photo of icebox and the prioritized features

As shown in the photo 6.2 a simple design with four bar buttons and only a few step of interaction layer were the final design.

Even though it was not the final design, this process made the implementation of the iPhone app a lot easier and cheaper in time compared to writing code as the first step. The main thoughts of the user experience were thereby settled before implementation.

### 6.1.3 From Mockup to iPhone App

The mockup and feature backlog gave guidelines for the implementation of the iPhone app. After the most important features were implemented the design and final polishing was added. A one-day session with the designer was enough to finish the overall design.

As shown in the before-after pictures on figure 6.3 the design clearly gave the app a more fresh and desirable look.

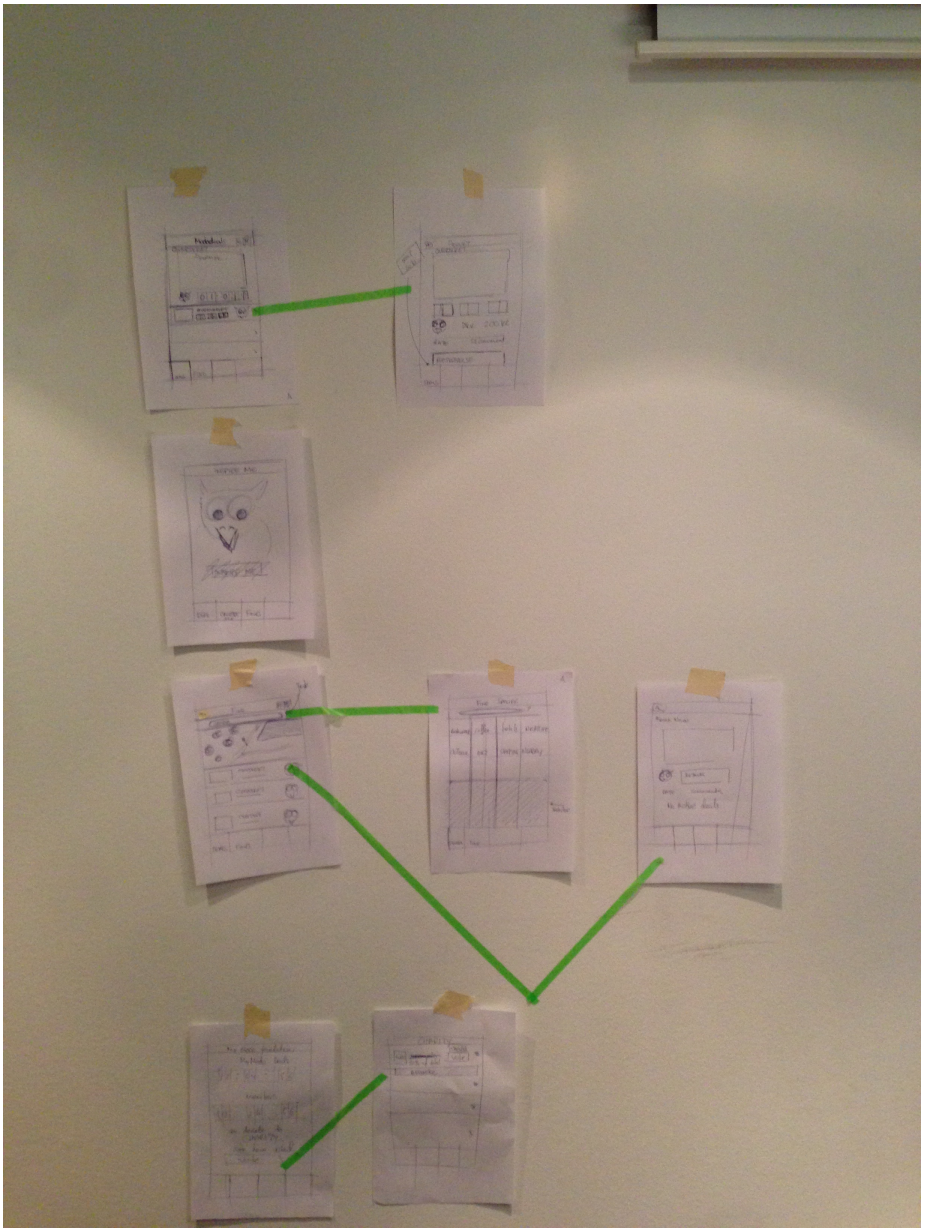
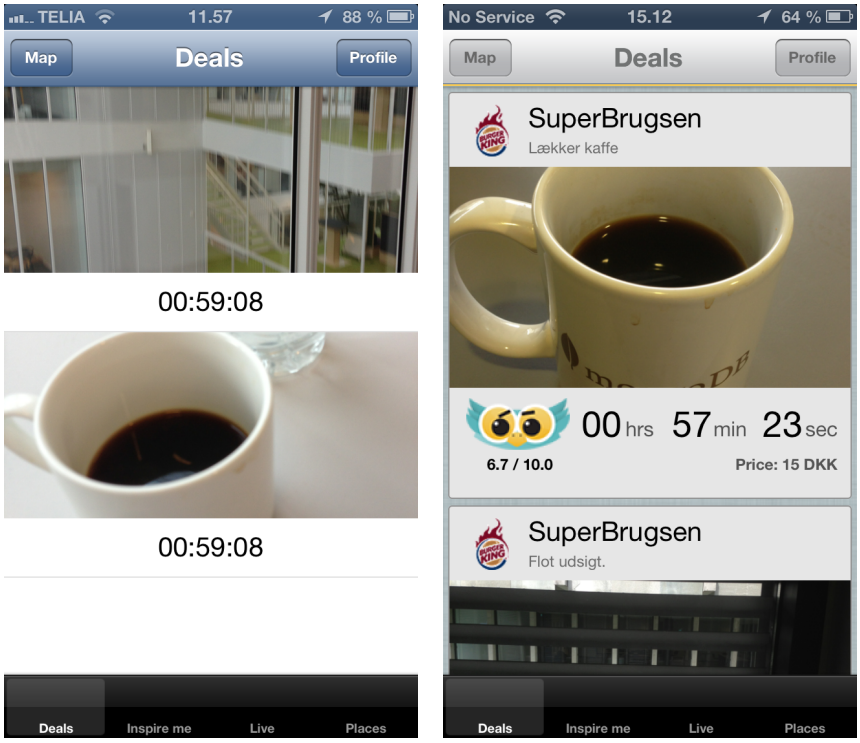


**Figure 6.2:** Photo of the first mockup




**(a)** The iPhone App before design **(b)** The iPhone App after design

**Figure 6.3:** A comparison of the iPhone App before and after the design was implemented

### 6.1.4 From iPhone App to App Store

This design gave the opportunity for user testing and the getting user feedback before submission to App Store is essential[1]. The approval process when submitting an app to Apples App Store is very strict and only by following Apples own guideline an app is accepted. [2] Apple always have their rights to remove or to not approve an app for any reason. The most restricted guidelines are: no adult content, the app has to be useful and unique and only serious development is accepted. The iPhone app developed in this project holds all the guideline even though an issue with in-app purchases may exist. Apple demands a cut of 30% of all payment done by the user when given digital content, which is what this iPhone app provides. Apple is not in charge of the payment, but digital

[1] See why user testing in 4.4.1
[2] https://developer.apple.com/appstore/resources/approval/guidelines.html

content is still provided throughout Apples services. Exactly how or if Apple demands a 30% cut is uncertain at given time, but this have to be considered before launching to App Store.

## 6.2 Project Management

The project was developed collaboratively and therefore needed a tool for managing the Kanban feature tracking board and the source code written. The online service PivotalTracker[3] was chosen as the feature tracking tool. For managing the source code and the code writing process, the version control system Git[4] was used together with the online source code service GitHub[5].

### 6.2.1 Feature Tracking

The feature tracking tool Pivotaltracker as seen in action on figure 6.4 was used as the Kanban visualized board. With PivotalTracker the developer instant have a overview of which features that should be implemented next and which features others in the team are working on. The team members are able to add features, bugs, predict complexity and even prioritize on what is most important. This tool can be use by none-technical people e.g. the salesman discovers an crucial bug, adds a story in Pivotaltracker and the developers are instantly informed.

### 6.2.2 Source Control

To be able to write and edit code in the same repository without worrying about what others are editing, the source control tool Git was used together with the repository server Github.

The code quality and maintainability in a repository can often be measured by the similarity of the developers coding style. To secure a more robust code repository in case of maintainability, code reviews was introduced in the process of pushing code to the main repository. GitHub provides a feature for this process called “Pull Requests”. The process of developing a feature and pushing it to the repository was as follow (and illustrated on figure 6.5)

[3] https://www.pivotaltracker.com/
[4] http://git-scm.com/
[5] https://github.com/

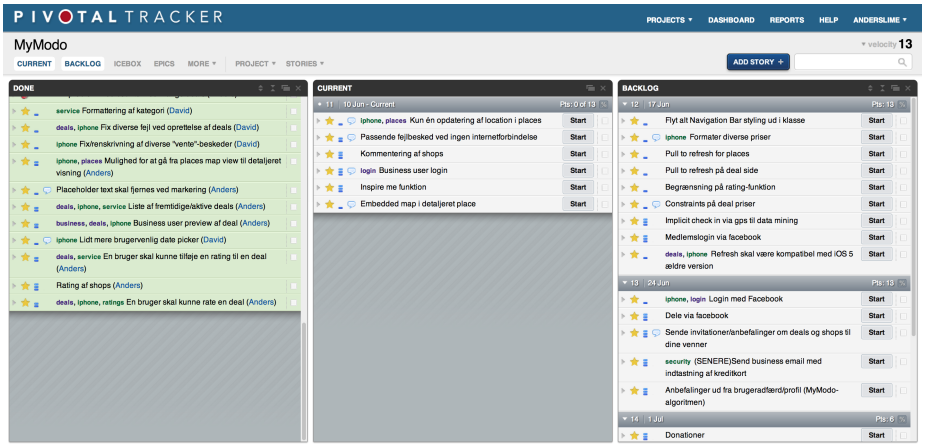



**Figure 6.4:** Screenshot of the Pivotaltracker page for the project

1. Create a branch for the given feature

2. Code the feature and divide the code into atomic commits

3. Push the given branch to GitHub and create a Pull Request for the branch

4. Another developer should perform code review and "merge" the branch into the master branch, when the code from the Pull Request is satisfactory.

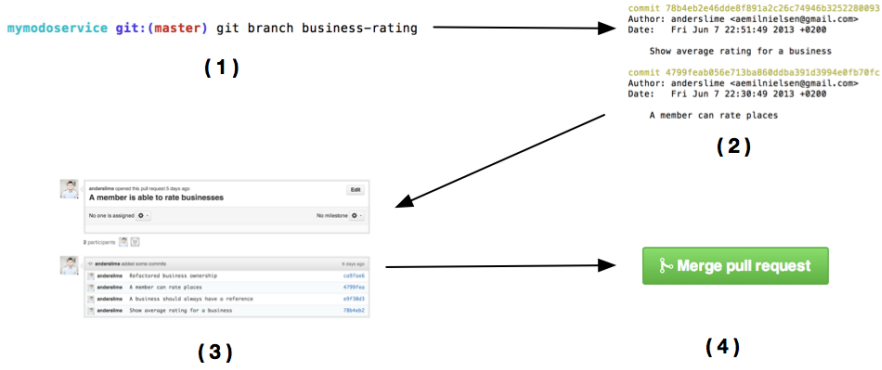



**Figure 6.5:** The flow of developing a feature and pushing it into the repository

## 6.3 Discussion

In the process of going from an idea to a mockup and feature backlog it was clear that everybody had different ideas of how the product should look. This emphasises the importance of taken everything abstract such as an idea and turning it into a concrete thing as a mockup such that everybody no matter the background are able to see how the product will be like. Because the effort of the other team members was on spare time basis, it was hard to coordinate meetups in order to continuously show the results of the product development. This sounds a bit like the waterfall process model, but since we were able to act as business owners ourselves, it was possible to continuously discuss and review the product development. In future development we will probably be working closer to the rest of the team such that everybody agree (including the customers).

The development of the system was done with Git (and GitHub), code reviews and Pivotal Tracker. Pivotal Tracker worked well for achieving an overview of what to do next. Git also worked great for source control, but there exists one problem between Git and Xcode that is remained unsolved - storyboard merge conflicts. The storyboard is represented as a single structured XML-like file, which means that when two developers are working on the same storyboard it would result in a merge-conflict that involved a file that has very little semantics and is therefore hard to merge manually. This problem was not solved fully, but two initiatives helped the process:

1. Create the UI in collaboration before implementing a feature. It is hard to predict all the storyboard changes of a feature, but by implementing the overall elements in the storyboard minimized the sizes of the merge-conflicts.
2. Create more storyboards. This project consists of 5 storyboard and we are are probably splitting the main one in the future. It does not take many lines of code to create a manual segue between two storyboards and it helps a lot on the Git-problem.

The code reviews done with GitHub also worked great and helped the learning process and understanding of each others code, but it was hard to be consistent about it due to the fact there only is one developer to perform code reviews. If only one developer is “at the office” it means that the implementation becomes one big pull request, which makes it hard to perform code review. It is optimal to have small pull requests that are easy to review, but this is hard to be consistent about, when there is only one reviewer.

Chapter 7

# Discussion

To two main principles we can take away from this project are:

- Focus on the core values of the business.
- Find a proper balance between time to market and requirements, when deciding technologies.

The development of a product should focus on core values instead of re-implementing custom libraries and infrastructures that is already available, it is more important to be best at what the business is unique at and offers to the customers. Examples in this project are the deployment infrastructure and image delivery. Instead of implementing a highly scalable image delivery service and web service infrastructure, we have focused at researching the market and current solutions and developed features for the iPhone app and resources for the web service API. The decision of starting with Cloud.dk was not the best, because Heroku provides a platform that has the robustness requirements such as scalability and availability. Also a VPS on Cloud.dk would need custom load balancers, networking setup and deployment recipes. But even though the decision of starting with Cloud.dk was bad, the knowledge from deploying to a VPS with Ruby on Rails and setting up a web server was valuable.

Time to market was an important factor of the product behind this project, but despite this we learned that it is important to consider the technologies behind the system. The technologies chosen for the system can decide the faith of the future system with relation to scalability and robustness. With a start-up that has a short time to market, it is often technologies that are familiar to the people behind the project. Even though the start-up behind this project has a time to market, it is still important that the product is able to scale in the future. By deeply considering the technologies of the product as done in this project, it was possible to design a scalable system that involved both well-known technologies, but also new technologies. We had to learn new technologies such as NoSQL and MongoDB, but the time spend on learning these technologies will work as an scalability and robustness investment for the future system.

Alternatively if the full focus were on time to market and the technologies just were chosen by what is optimal for the business by the knowledge at that time, a RDBMS and cross compiling framework such as PhoneGap could have been chosen. This would of course result in that more features were implemented on the iPhone, but the future system would be harder to scale and limits on gestures and animations would result in a worse user experience.

If the balance were too much on focusing on using and learning the most optimal technologies and implementing custom advanced systems, the project would not only fail at reaching the time to market. It would also end up with maintaining large systems that are not even important for the business.

Chapter 8

# Conclusion

With the careful chosen technologies that focus on scaling, user experience and robustness this mobile platform has become a reality. The interface is an iPhone app written in Objective-C with test proven user experience techniques. The web service is written in Ruby on Rails with a NoSQL database. A NoSQL database was chosen because of the natural ability to scale horizontally. The document-oriented NoSQL database MongoDB was chosen because of the ease of use and modelling structure. Cloud.dk is the cloud provider, but the scaling tests showed that with the PaaS Heroku the system was easier to scale and therefore Heroku will be used in the future. The database system will also be deployed to a DaaS such as MongoHQ. In future development, JRuby and JVM will be considered as technologies due to performance and scalability. An illustration of the future architecture is seen on figure 8.1.

With the technologies settled, the actual system was implemented in order to meet the requirements. The system decisions were made with focus on robustness, performance and statistically proven user experience techniques.

The process and the well-consider technology decisions were the foundation of a scalable system and the iPhone app with high user experience. After the final adjustments and the right business strategy were settled, this innovative iPhone app is ready for the App Store, where it will bring life and revenue back to the physical shops.

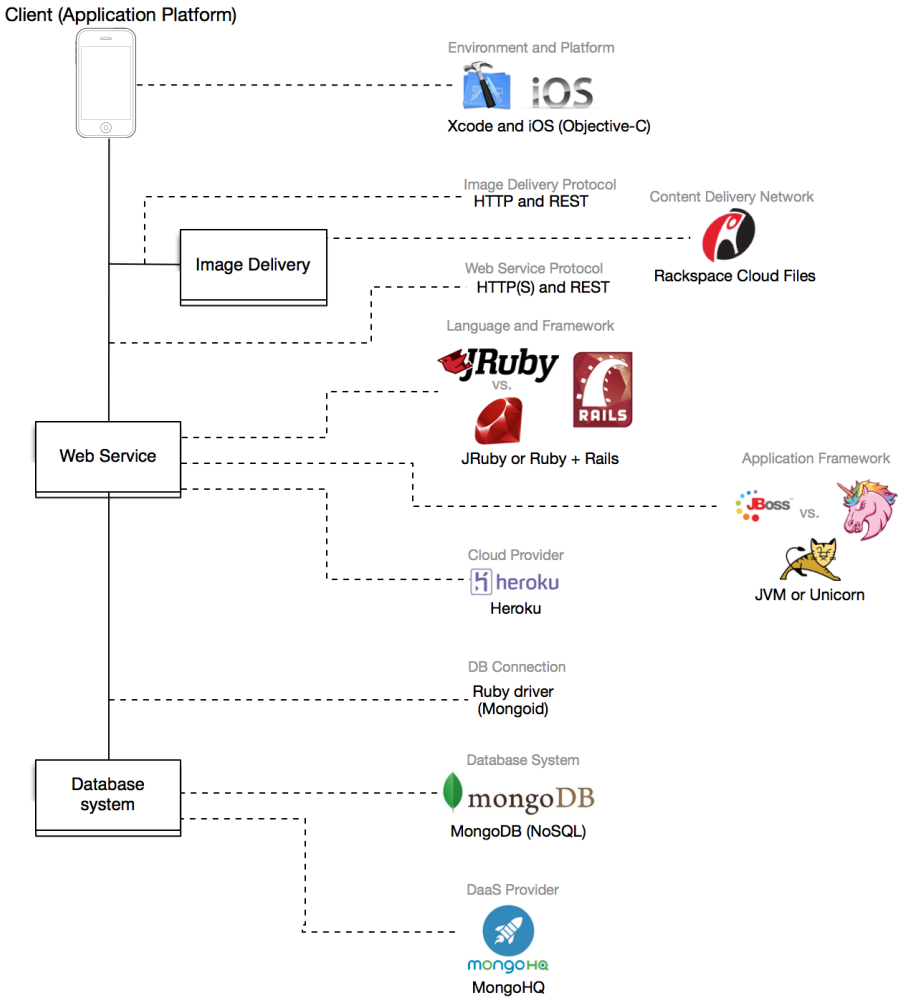



**Figure 8.1:** The future architecture of the system.

Appendix A

# Acronyms

**AWS** Amazon Web Services

**EC2** Amazon Elastic Compute Cloud

**DOM** Document-Oriented Modelling

**RDBMS** Relational Database Management System

**CDN** Content Delivery Network

**OSM** Open Street Maps

**ROA** Resource-Oriented Architecture

**GIL** Global Intepreter Lock

**AC** Availability and Consistency

**PA** Partition tolerance and Availability

**CP** Consistency and Partition tolerance

**DaaS** Database as a Service

**PaaS** Platform as a Service

**IaaS** Infrastructure as a Service

**VPS** Virtual Private Server

**LLVM** Low Level Virtual Machine

**JVM** Java Virtual Machine

Appendix B

# Jyllands Posten Article

Erhverv & Økonomi 24.05.13 kl. 03:00


# Salget i butikkerne er banket 10 år tilbage

AF Jens Erik Rasmussen

Forbrugerne har flere penge, men de bliver i lommerne. Salget af tøj og sko ligger underdrejet, men der er lys forude.

Salget i butikkerne skraber bunden. Købelysten fortsætter med at falde, og detailsalget er nu banket tilbage til samme niveau som for 10 år siden. Nedturen har været mere eller mindre konstant siden 2007, da forbrugsfesten toppede. Rutsjeturen betyder, at forretningerne på godt fem år har mistet knap 15 pct. af omsætningen.


Detailhandlen havde håbet, at salget i april ville vise fremgang, fordi marts for mange brancher i detailhandlen var en ualmindelig dårlig måned. Men genopretningen udeblev, og salget i april var 1,5 pct. lavere end i samme måned for et år siden renset for prisstigninger.

»Detailsalget er blandt de værst ramte brancher i dansk økonomi. Salget i butikkerne er faldet betydeligt mere end det samlede private forbrug, som i de seneste år stort set har været uændret,« siger Jacob Graven, cheføkonom i Sydbank.

## Psykologiske årsager

Han vurderer, at det især er psykologiske årsager, som afholder danskerne fra at bruge penge. De fleste forbrugere har nemlig fået flere penge mellem hænderne i de seneste år på grund af lave renter og skattelettelser. Samtidig er der udbetalt milliarder i efterlønspenge.

[citat-1]

»I år vil reallønnen sandsynligvis stige for de fleste danskere, da inflationen er faldet markant, men foreløbig har det ikke fået danskerne til at bruge flere penge,« konstaterer Jacob Graven.

Han fremhæver, at usikkerhed om den økonomiske situation, et svagt boligmarked og frygten for at miste

jobbet får danskerne til at holde igen. Politiske reformer om dagpenge, kontanthjælp og efterløn får også forbrugerne til at tøve.

Salget af tøj og sko har i årets første måneder ligget underdrejet i forhold til sidste år. Men branchen tror på en stabilisering.

»Marts var en ekstremt dårlig måned, og salget af sko dykkede med 25 pct., mens omsætningen af tøj faldt med 7 pct. I april har der været fremgang i forhold til sidste år,« siger Jens Birkeholm, direktør i Dansk Detail.

Detailsalget i USA steg overraskende

## Kuldegrader koster

Det er først og fremmest det kolde vejr, som har kostet kunder. I skobutikkerne er vinterfodtøjet på udsalg, men forbrugerne tøver, fordi de venter på et mildere klima. Kuldegraderne betyder omvendt, at forårsvarerne bliver stående på hylderne. Samme tendens ses i tøjbutikkerne, omend det er mindre tydeligt.

I april har sko og tøj vundet noget af det tabte tilbage, men salget for årets første fire måneder ligger fortsat 3 pct. under niveauet fra sidste år.

Tal fra Danmarks Statistik viser, at nedgangen i forbruget rammer meget forskelligt. Siden 2010 er salget faldet markant hos møbelhandlere, tæppeforretninger, byggemarkeder og farvehandlere.

»Brancher, som er afhængige af udviklingen på boligmarkedet, har det svært. Salget af boliger går trægt, og det er ofte i forbindelse med flytning, at forbrugerne køber nyt,« siger Jens Birkeholm.

Også inden for dagligvarer er der sket en markant udvikling. Discountforretninger har på tre år øget omsætningen med 29 pct. Det ekstra salg er hentet hos supermarkeder, købmænd, kiosker og frugt og grønt-forretningerne. Udviklingen ventes at fortsætte, da lukkeloven har givet medvind til kæder som Netto, Rema 1000 og Lidl.

»Det er et klassisk eksempel på, at danskerne er tilbudsjægere, som går efter de laveste priser,« siger Jens Birkeholm

Forbrugere tøjler tristessen (2)

Appendix C

# Scalability Tests

The results from the tests performed in section 4.4.2.

# Cloud.dk Micro

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking v1.limecode.dk (be patient)


Server Software:        nginx/1.2.7
Server Hostname:        v1.limecode.dk
Server Port:            80

Document Path:          /
Document Length:        49 bytes

Concurrency Level:      50
Time taken for tests:   129.523 seconds
Complete requests:      10000
Failed requests:        86
   (Connect: 0, Receive: 43, Length: 43, Exceptions: 0)
Write errors:           0
Total transferred:      4241682 bytes
HTML transferred:       487893 bytes
Requests per second:    77.21 [#/sec] (mean)
Time per request:       647.616 [ms] (mean)
Time per request:       12.952 [ms] (mean, across all concurrent requests)
Transfer rate:          31.98 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   70 1141.2     10   68381
Processing:     6  434 4954.0    111   76120
Waiting:        5  108  61.0    110     476
Total:         10  503 5078.2    123   76120

Percentage of the requests served within a certain time (ms)
  50%    123
  66%    129
  75%    149
  80%    159
  90%    191
  95%    246
  98%    313
  99%   2204
 100%  76120 (longest request)
```

# Heroku 1 dyno

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking mymodo.herokuapp.com (be patient)

Server Software:
Server Hostname:        mymodo.herokuapp.com
Server Port:            80

Document Path:          /v1
Document Length:        49 bytes

Concurrency Level:      50
Time taken for tests:   84.721 seconds
Complete requests:      10000
Failed requests:        114
   (Connect: 0, Receive: 69, Length: 45, Exceptions: 0)
Write errors:           20
Non-2xx responses:      2
Total transferred:      4040623 bytes
HTML transferred:       488997 bytes
Requests per second:    118.04 [#/sec] (mean)
Time per request:       423.603 [ms] (mean)
Time per request:       8.472 [ms] (mean, across all concurrent requests)
Transfer rate:          46.58 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      100  370 354.3    285   14510
Processing:     0   53  84.0     39    4764
Waiting:        0   45  48.9     34     636
Total:        215  423 376.8    334   14510

Percentage of the requests served within a certain time (ms)
  50%    334
  66%    386
  75%    431
  80%    468
  90%    672
  95%    883
  98%   1151
  99%   1394
 100%  14510 (longest request)
```

# Heroku 2 dynos

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking mymodo.herokuapp.com (be patient)

Server Software:
Server Hostname:        mymodo.herokuapp.com
Server Port:            80

Document Path:          /v1
Document Length:        49 bytes

Concurrency Level:      50
Time taken for tests:   77.935 seconds
Complete requests:      10000
Failed requests:        22
   (Connect: 0, Receive: 18, Length: 4, Exceptions: 0)
Write errors:           6
Total transferred:      4062945 bytes
HTML transferred:       490049 bytes
Requests per second:    128.31 [#/sec] (mean)
Time per request:       389.677 [ms] (mean)
Time per request:       7.794 [ms] (mean, across all concurrent requests)
Transfer rate:          50.91 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      100  345 195.0    263    1942
Processing:     0   44  47.2     30     562
Waiting:        0   41  46.3     27     561
Total:        210  389 208.7    314    2019

Percentage of the requests served within a certain time (ms)
  50%    314
  66%    371
  75%    424
  80%    470
  90%    639
  95%    837
  98%   1090
  99%   1261
 100%   2019 (longest request)
```

# Heroku 3 dynos

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking mymodo.herokuapp.com (be patient)

Server Software:
Server Hostname:        mymodo.herokuapp.com
Server Port:            80

Document Path:          /v1
Document Length:        49 bytes

Concurrency Level:      50
Time taken for tests:   71.984 seconds
Complete requests:      10000
Failed requests:        20
   (Connect: 0, Receive: 11, Length: 9, Exceptions: 0)
Write errors:           6
Total transferred:      4053236 bytes
HTML transferred:       489608 bytes
Requests per second:    138.92 [#/sec] (mean)
Time per request:       359.918 [ms] (mean)
Time per request:       7.198 [ms] (mean, across all concurrent requests)
Transfer rate:          54.99 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      101  322 182.3    255    1803
Processing:     0   37  42.3     25     790
Waiting:        0   32  41.3     21     789
Total:        215  359 191.1    294    1830

Percentage of the requests served within a certain time (ms)
  50%    294
  66%    325
  75%    357
  80%    387
  90%    555
  95%    835
  98%   1036
  99%   1166
 100%   1830 (longest request)
```

# Heroku 4 dynos

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking mymodo.herokuapp.com (be patient)

Server Software:
Server Hostname:        mymodo.herokuapp.com
Server Port:            80

Document Path:          /v1
Document Length:        49 bytes

Concurrency Level:      50
Time taken for tests:   67.706 seconds
Complete requests:      10000
Failed requests:        29
   (Connect: 0, Receive: 21, Length: 8, Exceptions: 0)
Write errors:           8
Total transferred:      4063105 bytes
HTML transferred:       489657 bytes
Requests per second:    147.70 [#/sec] (mean)
Time per request:       338.528 [ms] (mean)
Time per request:       6.771 [ms] (mean, across all concurrent requests)
Transfer rate:          58.60 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       99  307 176.4    249    2311
Processing:     0   30  38.3     19     968
Waiting:        0   26  34.5     15     856
Total:        215  337 187.1    277    2411

Percentage of the requests served within a certain time (ms)
  50%    277
  66%    302
  75%    327
  80%    354
  90%    483
  95%    703
  98%    986
  99%   1224
 100%   2411 (longest request)
```

# Heroku 5 dynos

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking mymodo.herokuapp.com (be patient)

Server Software:
Server Hostname:        mymodo.herokuapp.com
Server Port:            80

Document Path:          /v1
Document Length:        49 bytes

Concurrency Level:      50
Time taken for tests:   71.508 seconds
Complete requests:      10000
Failed requests:        33
   (Connect: 0, Receive: 19, Length: 14, Exceptions: 0)
Write errors:           7
Total transferred:      4051621 bytes
HTML transferred:       489461 bytes
Requests per second:    139.84 [#/sec] (mean)
Time per request:       357.539 [ms] (mean)
Time per request:       7.151 [ms] (mean, across all concurrent requests)
Transfer rate:          55.33 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      100  327 218.1    254    3516
Processing:     0   30  38.8     19    1215
Waiting:        0   25  34.0     14     776
Total:        208  356 228.9    282    3811

Percentage of the requests served within a certain time (ms)
  50%    282
  66%    312
  75%    346
  80%    377
  90%    557
  95%    799
  98%   1039
  99%   1407
 100%   3811 (longest request)
```

# Heroku 6 dynos

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking mymodo.herokuapp.com (be patient)

Server Software:
Server Hostname:        mymodo.herokuapp.com
Server Port:            80

Document Path:          /v1
Document Length:        49 bytes

Concurrency Level:      50
Time taken for tests:   57.192 seconds
Complete requests:      10000
Failed requests:        6
   (Connect: 0, Receive: 3, Length: 3, Exceptions: 0)
Write errors:           3
Total transferred:      4053635 bytes
HTML transferred:       489951 bytes
Requests per second:    174.85 [#/sec] (mean)
Time per request:       285.959 [ms] (mean)
Time per request:       5.719 [ms] (mean, across all concurrent requests)
Transfer rate:          69.22 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      102  257  93.1    230    1275
Processing:     0   29  26.9     19     425
Waiting:        0   25  25.6     15     424
Total:        214  285  98.3    255    1275

Percentage of the requests served within a certain time (ms)
  50%    255
  66%    272
  75%    287
  80%    300
  90%    348
  95%    450
  98%    631
  99%    760
 100%   1275 (longest request)
```

Appendix D

# User Tests

Download links for all the user test video sessions.

1. Nikolai Jensen
2. Mads 1. video
3. Mads 2. video
4. Mads 3. video
5. Lars Sjælland 1. video
6. Lars Sjælland 2. video
7. Lars Sjælland 3. video
8. Lars Sjælland 4. video
9. Lars Sjælland 5. video
10. Andreas Graulund 1. video
11. Andreas Graulund 2. video
12. Andreas Graulund 3. video

13. Andreas Graulund 4. video
14. Andreas Graulund 5. video
15. Andreas Graulund 6. video
16. Andreas Graulund 7. video
17. Andreas Graulund 8. video

# Bibliography


[10g13] Inc. 10gen, June 2013. http://docs.mongodb.org/manual/core/data-modeling/, [seen 14. June 2013].

[ama07] Dynamo: Amazon's highly available key-value store. 2007. http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/decandia07dynamo.pdf.

[BF04] Matt Bishop and Deborah Frincke. Teaching robust programming. *IEEE Security & Privacy*, 2004.

[Bre00] Eric A. Brewer. Towards robust distributed systems. 2000. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[CDK05] Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[CW] GitHub Chris Wanstrath. http://rack.github.io/, [seen 21 June 2013].

[DE13] DB-Engines. Db-engines ranking, June 2013. http://db-engines.com/en/ranking, [seen 14. June 2013].

[Ene] Tom Enebo. Jruby: You've got java in my ruby. http://www.infoq.com/presentations/enebo-jruby.

[For99] The Internet Engineering Task Force, June 1999. http://www.ietf.org/rfc/rfc2617.txt, [seen 24. June 2013].

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *ACM SIGACT News*, page 2002, 2002.

[goo06] Bigtable: A distributed storage system for structured data. 2006. http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//archive/bigtable-osdi06.pdf.

[JN13] Raluca Budiu Jakob Nielsen. *Mobile Usability*. New Riders, Berkeley, CA 94710, 2013.

[Kut13] Joe Kutner. *Deploying with JRuby*. The Pragramatic Programmers, LLC., 2013.

[Lie11] Chad Lieber, Ethan & Syverson. Online vs. offline competition. January 2011. Relevant information at table 1, page 38.

[Ras13] Jens Erik Rasmussen. Salget i butikkerne er banket 10 aar tilbage. *Jyllands Posten, Erhverv & Oekonomi*, May 2013. Is not available offline and therefore attached as appendix.

[RR07] Leonard Richardson and Sam Ruby. *Restful web services*. O'Reilly, first edition, 2007.

[rt] ruby toolbox.com. https://www.ruby-toolbox.com/categories/web_servers, [seen 27 June 2013].

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[weba] http://www.cloud.dk, [seen 28 June 2013].

[webb] http://aws.amazon.com/ec2/, [seen 28 June 2013].

[webc] http://heroku.com.

[webd] http://rspec.info, [seen 28 June].

[webe] http://httpd.apache.org/docs/2.2/programs/ab.html, [seen 23 June 2013].

[webf] https://devcenter.heroku.com/articles/dyno-size, [seen 28 June 2013].

[webg] https://blog.heroku.com/archives/2013/4/5/2x-dynos-beta, [seen 29 June 2013].