

# Helvetas Horticulture System

## Bachelor Thesis

Department of Computer Science  
University of Applied Science Rapperswil

Fall Term, 2015

Author(s): Rico Beti, Simon Zingg  
Advisor: Prof. Frank Koch  
Project Partner: HELVETAS Nepal  
External Co-Examiner: Mr. Stephan Meier  
Internal Co-Examiner: Prof. Dr. Eduard Glatz

Bachelor Thesis  
Helvetas Horticulture System

RICO BETI (rbeti@hsr.ch)  
SIMON ZINGG (szingg@hsr.ch)

December 18<sup>th</sup>, 2015

## 1. Abstract

This bachelor thesis lays the foundation of a new project with HELVETAS Nepal. As part of their “Food Security and Nutrition” undertaking, HELVETAS is focusing on basic human needs such as clean drinking water, food, education, and peace. Specifically with the help of this bachelor thesis, HELVETAS tries to streamline their support for local Nepali farmers that live far away from larger cities or villages and therefore have no access to agricultural assistance.

The main idea behind the “HELVETAS Horticulture System” is the distribution of Android phones to local farmers and the creation of a ticketing system that allows these farmers to receive help online without the need of an expert actually visiting the remote villages. This approach not only saves precious time but also allows HELVETAS to dispatch their resources appropriately and effectively. The system brings great analytical benefits as well. HELVETAS analysts are able to use the collected data for situation assessments and may detect recurring problems, such as a spreading pest infestation in a specific area.

While the system is first and foremost being developed for horticulture application under Open Source GPL3 terms, the core follows a strictly generic approach which allows the system domain to be expanded to different areas such as health care.

Within the boundaries of this bachelor thesis lie the following key tasks:

- In-depth analysis of the problem and tasks at hand.
- Establishment of concepts and appropriate solutions.
- Development of a system prototype for further development by HELVETAS.
- Extensive documentation of code, system administration and system usage.

In this document, the problem specifics, requirements, key tasks, and established solutions and conclusions will be described in detail. The thesis aims to explain concepts and system architecture in a way that is understandable and straightforward, so that the system may be continuously developed by HELVETAS software developers in the near future.

## Document Changes

---

<b>Date</b>	<b>Version</b>	<b>Change</b>	<b>Author(s)</b>
2015-09-17	1.0	Initial Version	Rico Beti, Simon Zingg
2015-12-18	1.1	Final Version	Rico Beti, Simon Zingg

---

## Statement of Authorship

We hereby declare,

- that this project, including thesis, all other documents, and source code, are results of our own work and that we have not received any assistance other than what has been specified in the project specification, or what has been agreed upon in writing with the examiner,
- that we have referenced all source material according to generally accepted scientific citation guidelines,
- and that we have not used any resources protected under copyright law without appropriate permission.

Jona, 18<sup>th</sup> of December 2015



Rico Beti



Simon Zingg

## Table of Contents

<b>1</b>	<b>Abstract</b> .....	<b>3</b>
<b>2</b>	<b>Management Summary</b> .....	<b>10</b>
<b>3</b>	<b>Domain Analysis</b> .....	<b>12</b>
3.1	Domain Model .....	12
3.2	Domain Terminology .....	14
3.2.1	Regular Farmers .....	14
3.2.2	Lead Farmers a.k.a. Users .....	14
3.2.3	Experts .....	15
3.2.4	Administrators .....	15
3.2.5	Android Application .....	15
3.2.6	Administration Client .....	15
3.2.7	Server .....	15
3.2.8	Ticket System .....	16
3.2.9	Thread a.k.a. Ticket .....	16
3.2.10	Post .....	16
3.2.11	Queue .....	16
3.2.12	Database .....	16
3.2.13	System Base & Knowledge Base .....	17
3.2.14	Alert .....	17
3.2.15	Service .....	17
3.2.16	News Feed .....	17
3.3	Types of Users .....	18
3.4	Environment .....	18
3.5	Competing Software .....	18
3.6	Extensibility to Other Domains .....	18
<b>4</b>	<b>Requirement Analysis</b> .....	<b>19</b>
4.1	Functional Requirements .....	19
4.1.1	Use Case Diagram .....	20
4.1.2	FR: Authentication & Authorization .....	21
4.1.3	FR: Queries, Ticketing System .....	23
4.1.4	FR: Knowledge Base .....	26
4.1.5	FR: Services .....	27
4.1.6	FR: Alerts .....	28
4.1.7	FR: Analysis - Data Mining .....	28
4.2	Nonfunctional Requirements .....	29
4.2.1	Technologies .....	29

---

4.2.2	Open Source.....	29
4.2.3	Data Rates & Data Volume.....	30
4.2.4	Usability .....	31
4.2.5	User Experience.....	32
4.2.6	Scalability .....	33
4.2.7	Internationalization.....	35
4.2.8	Generic System.....	36
4.2.9	Documentation .....	36
<b>5</b>	<b>Architecture .....</b>	<b>37</b>
5.1	System Overview.....	37
5.2	Deployment Diagram.....	38
5.3	Architectural Decisions .....	39
5.3.1	ASP.NET .....	39
5.3.2	Reasons for Building Native Applications .....	39
5.3.3	Communication to the Client.....	41
5.3.4	Server Layers: Why Two Layers & Why no DAL? .....	42
5.3.5	Why Windows Forms was Chosen instead of WPF .....	43
5.4	Server.....	43
5.4.1	Presentation Layer.....	43
5.4.2	Business Logic Layer.....	44
5.4.3	Dependencies.....	47
5.5	HHS.Common Project .....	48
5.6	Administration Client .....	50
5.6.1	Dependencies.....	52
5.7	Android Application.....	52
5.7.1	Activity Lifecycle Management .....	53
5.7.2	General App Architecture .....	54
5.7.3	Local Database / Data Model.....	54
5.7.4	Database Diagram .....	55
5.7.5	ORM: Active Android.....	55
5.7.6	Offline Mode & Synchronization .....	56
5.8	SMS Gateway .....	57
5.9	Database .....	58
5.9.1	Use of GUIDs .....	60
5.9.2	Table Names .....	60
5.9.3	Field Names .....	60
5.9.4	Field Types.....	60
5.9.5	Language Identifiers.....	61

---

<b>6</b>	<b>Concepts</b> .....	<b>62</b>
6.1	Ticket System .....	62
6.1.1	Thread Queues & Thread Filters .....	63
6.2	Location Determination .....	65
6.3	Knowledge Base .....	66
6.3.1	KB Structure.....	66
6.3.2	Tags.....	66
6.3.3	Search Mechanism .....	67
6.3.4	Accuracy .....	67
6.3.5	Localization.....	68
6.3.6	Concurrency .....	69
6.3.7	Upload Issues with Large Files .....	69
6.4	News Feed - Services .....	69
6.4.1	Weather Forecast .....	70
6.4.2	Market Price .....	72
6.4.3	Quartz.NET – Job Scheduler .....	72
6.4.4	Extensibility .....	73
<b>7</b>	<b>Results</b> .....	<b>74</b>
7.1	Achievements .....	74
7.1.1	Achievement: Server .....	74
7.1.2	Achievement: Administration Client .....	75
7.1.3	Achievement: Android Application .....	75
7.2	Open Tasks.....	76
7.2.1	Server .....	76
7.2.2	Administration Client .....	76
7.2.3	Android Client .....	76
7.2.4	Use Cases.....	77
7.3	From Development to Production.....	77
7.3.1	Server .....	77
7.3.2	Admin Client.....	77
7.3.3	Android Client .....	78
7.4	Lessons Learned .....	78
<b>8</b>	<b>Project Management</b> .....	<b>79</b>
8.1	Time Management .....	79
8.2	Division of work .....	81
8.3	Communication .....	81
8.4	Risk Management.....	82



<b>9 Documentation</b> .....	<b>84</b>
9.1 Doxygen .....	84
9.2 Server Online API Documentation.....	84
<b>List of Figures</b> .....	<b>85</b>
<b>References</b> .....	<b>86</b>
<b>Appendices</b> .....	<b>87</b>
A Attachments .....	87
B Licenses .....	88

## 2. Management Summary

The main goal of this bachelor thesis is the design and development of a ticket system that is primarily going to be used for horticulture application by HELVETAS Nepal. The system will allow HELVETAS Nepal to improve the effectiveness of their humanitarian efforts in helping local farmers. Using the system, HELVETAS Nepal will be able to more precisely allocate available resources and channel funds to places where they are needed most.

### Motivation

Farmers in Nepal are scattered and often live in remote communities far away from other villages and cities. They generally do not have easy access to markets and their products are of little interest to local businesses. These circumstances make it incredibly difficult for HELVETAS Nepal to provide help effectively and efficiently as personal visits are extremely time consuming.

HELVETAS Nepal realized that mobile network coverage in Nepal is quite acceptable, even in remote areas. As mobile phones are getting cheaper and more affordable, HELVETAS Nepal saw an opportunity for improvement.

### Approach & Technologies

The most important parts of this bachelor thesis are the analysis of the current situation, the establishment of concepts, and the development of a prototype that reflects parts of the required functionality.

With larger goals in mind, the system shall be designed with a generic core so that it may be extended onto other domains, such as health care and education.

As this will be a distributed system that is required to operate on cost-effective low-end hardware with efficient communication, several different technologies will have to be addressed and incorporated, which will be elaborated in detail in this report.

## Results

The results of this bachelor thesis include an in-depth analysis of the problem domain, the establishment of a sophisticated solution that is meeting the requirements of HELVETAS Nepal, and a working prototype system that is ready for further development.

As for the prototype, the following three artifacts have been developed and delivered:

- A server back end application that provides most of the requested functionality.
- A Microsoft Windows Client application intended to be used by HELVETAS Nepal employees and volunteers for providing assistance to farmers.
- An Android application that is to be distributed among farmers in Nepal which allows them to request help and ask questions.

## Future Endeavors

The HELVETAS Horticulture System is a large and complex software project that can not be completed within the duration of this bachelor thesis. While the server side is quite advanced and provides most of the functionality required by HELVETAS Nepal, both the Windows Client and the Android Application prototypes will have to be developed further before they can be deployed into a productive environment.

### **3. Domain Analysis**

The following sections provide information about the domain of the HELVETAS Horticulture System and try to elaborate the system's interoperability between its components and its users.

#### **3.1. Domain Model**

The HELVETAS Horticulture System is quite complex by nature as it involves multiple different actors and is run in a diverse system landscape. In order to enhance the understandability of the system and its components, the Domain Model (see Figure 1) will be introduced in the very beginning of this section and serves as an overview over the project. The Domain Model is introduced early in order to help clarifying structure, dependencies and terminology of different elements that play vital roles within this particular system. The Domain Model will be elaborated in detail throughout the course of this section.

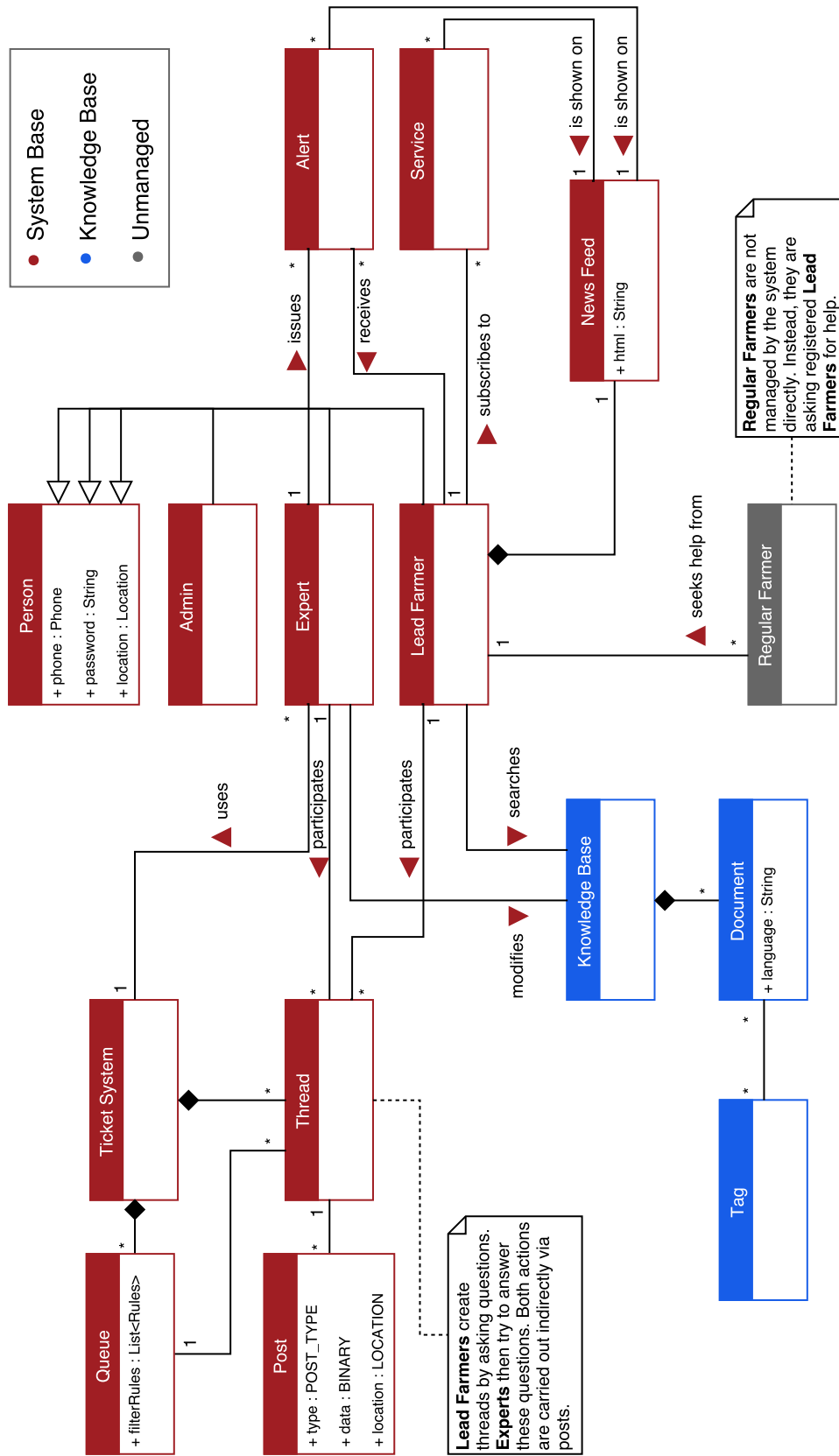


Figure 1: Domain Model

## 3.2. Domain Terminology

Several distinct and specific terms shall be introduced in this section that will be used throughout the rest of the paper. These terms refer to specific components or actors that are involved in the system. This chapter is also meant to serve as a reference guide for introducing new developers into the project.

*Relevant terms will start with a capital letter in this section to increase recognition.*

### 3.2.1. Regular Farmers

Regular Farmers are the main target audience of the system's domain and will receive the most benefit. However, these farmers are not registered within the system directly but instead ask Lead Farmers (Section 3.2.2) for help. Regular Farmers are generally not literate and therefore rely on the assistance of Lead Farmers. If Regular Farmers are in need of Expert (Section 3.2.3) assistance, they will ask a nearby Lead Farmer to send a request for help via the Lead Farmer's phone.

### 3.2.2. Lead Farmers a.k.a. Users

Lead Farmers are farmers that are equipped with Android phones and use the Android app (Section 3.2.5) that is also part of this project. Each Lead Farmer is registered in the system and owns an account with user name (phone number) and password. Lead Farmers are going to meet Regular Farmers personally on a regular basis and try to help them wherever they can by either delivering personal assistance, by searching information in the system's knowledge base, or by asking Experts for advice. Once Lead Farmers have found or received answers, they will forward the information to Regular Farmers. If needed, Lead Farmers are able to ask Experts to come to their village. HELVETAS assesses the necessity and dispatches Experts for personal visits if required.

Due to the generic nature of this project, it has been decided to simply refer to Lead Farmers as *Users*. The term Lead Farmer is entirely specific to the problem domain of HELVETAS. Whenever a more generic system component is referred, the term *User* will be used.

### **3.2.3. Experts**

Experts mainly use the Administration Client (Section 3.2.6) for processing Threads within the Ticket System (Section 4.1.3). They are providing help to the Users of the system by solving their requests. Experts also search and maintain the Knowledge Base (Section 3.2.13) and are responsible for the activation of user accounts.

### **3.2.4. Administrators**

Administrators maintain the system and its database. They do not directly engage with the content and deliver services from a technical point of view only.

### **3.2.5. Android Application**

The Android Application, or “app”, is front-end of the project and is used by the Users (Lead Farmers) of the system. It allows users to issue requests, call for help, access the Knowledge Base, subscribe to Services, receive Alerts, etc.

### **3.2.6. Administration Client**

The Administration Software is mainly used by Experts. It is developed for Microsoft Windows and runs on both Desktop and Laptop Systems. It features a graphical user interface in form of a “Ticket System”. Experts will get an overview of open Threads (Section 3.2.10) that can be filtered. They can engage in written conversation with the issuer of the Thread. Once a Thread has been solved and completed, it will be marked as such and archived for future reference. Experts may search for previous Threads. In addition, the Administration Software provides access to the Knowledge Base and allows the creation of new entries.

HELVETAS informed us that Administration Clients will have access to a good Internet connection (generally WLAN). This allows us to send and receive data more frequently and liberally, compared to the Android app.

### **3.2.7. Server**

The Server is the main backbone of the project and the central hub for the Administration Software and the Android App. It connects Users and Experts via the ticket system. The Server manages the System Base and Knowledge Base and handles requests via the system’s RESTful web service endpoint.

### 3.2.8. Ticket System

The Ticket System consists of all Threads, Posts and Queues that are managed by the Server. Experts use the Administration Client and Users use the Android app in order to interact with the system by reading and creating Posts.

### 3.2.9. Thread a.k.a. Ticket

A Thread is usually created by a User by submitting a question to the system. It consists of individual posts and represents a conversation between Experts and the User. Threads can be searched by Experts using the Administration Software.

*The term Ticket is equivalent to the term Thread, though Thread is preferred.*

### 3.2.10. Post

A Post is always part of a Thread and is issued by either a User or an Expert. Each Post has a specific type, e.g. Text, Voice, Image. A Post may be flagged as an internal note that is only visible to Experts.

### 3.2.11. Queue

Queues present a way of intelligently filtering threads based on specific filter criteria that can be defined by Experts. A good example is the *Proximity Queue* which has been implemented in the prototype. It allows Experts to filter Threads based on physical distance to the User that has issued the thread.

### 3.2.12. Database

The Database unifies System Base and Knowledge Base (Section 3.2.13) and refers to all information on the Server that is required for the system to fully operate.

$$Database = System\ Base \cup Knowledge\ Base$$



### 3.2.13. System Base & Knowledge Base

The Database is divided into two logically different parts, namely System Base and Knowledge Base. Both parts, however, normally run on the same physical system.

**System Base** refers to the *operational part* of the database, i.e. user records, thread and post data, etc. It *does not* include documents and other information from the Knowledge Base.

**Knowledge Base** is the name of the *informative part* of the database where all manually prepared information is stored. It is separated from the System Base and its content is entirely purpose specific. For this project, the Knowledge Base contains vital data that is aimed specifically at horticulture in Nepal.

*The listing below shows several aspects that are managed by the components.*

System Base	Knowledge Base
Includes <ul style="list-style-type: none"> <li>• User Management</li> <li>• Threads &amp; Posts</li> <li>• Alerts &amp; Services</li> </ul>	Includes <ul style="list-style-type: none"> <li>• Case Studies</li> <li>• Best Practices</li> <li>• How-To Guides &amp; FAQ</li> </ul>

### 3.2.14. Alert

Alerts are always issued either by an Expert or automatically by the system. The system tries to forward alerts to Users as fast as possible using SMS. Possible reasons to issue Alerts are storm warnings, plague spread, or other important information.

### 3.2.15. Service

Users can subscribe and unsubscribe to and from different services. Possible services include weather forecasts, information related to certain crops, market prices, etc.

### 3.2.16. News Feed

Services and alerts are sent directly to a User's Android app and are posted on the News Feed. News will only be received for those services for which the user is currently subscribed.

### 3.3. Types of Users

As mentioned before, three main categories of users will be accessing the system: *Users*, *Experts*, and *Administrators*. It is important to understand that these types of users come with varying degrees of previous knowledge. For instance, it is to be expected that Lead Farmers do not necessary know how to operate a phone. This circumstance must be considered and reflected in the design of the user interfaces.

### 3.4. Environment

A challenge that should not be underestimated is the rich diversity and heterogeneity of the domain's system landscape. To add to the difficulty of the system's distributed nature, unreliable Internet connectivity must be addressed at an early stage of development.

HELVETAS is primarily running Microsoft Windows technologies on their servers, desktop computers, and laptops. The system is required to handle constant hardware and software changes.

### 3.5. Competing Software

While there are a large number of different ticketing systems and customer support systems available as both open source and proprietary projects, none of them seemed to meet the requirements that were issued by HELVETAS.

Projects, such as *osTicket*<sup>1</sup> and *Zendesk*<sup>2</sup>, primarily focus on the system's back end, i.e. the Expert's point of view, and do not propose customizable solutions for the client side that could be adjusted to fit the specific needs of this project's primary target audience.

### 3.6. Extensibility to Other Domains

On of the basic requirement for this project is keeping extensibility in mind during the design process. While primarily being developed as a system for horticulture application, the system must allow for a simple adoption to other domains such as health care and education.

---

<sup>1</sup><http://osticket.com/>

<sup>2</sup><https://www.zendesk.com/>

## 4. Requirement Analysis

This section describes the functional and nonfunctional requirements of this project and provides a detailed listing of use cases.

### 4.1. Functional Requirements

The requirements that were initially provided by HELVETAS were little more than loosely structured and incoherent thoughts which conveyed a vague and uncertain image of what they imagined the horticulture system to be.

*It is important to understand that the concrete elaboration of the functional requirements is one of our work results of the first two weeks.*

We have realized at the start of the project that it is a substantial challenge to ensure that both parties (us at HSR and HELVETAS in Nepal) are talking about the same things due to different technical and cultural background.

In order to improve the communication and reduce chances of misunderstanding between HELVETAS Nepal and the development team in Switzerland, we have decided to specify the functional requirements in the form of Use Cases.

A Use Case (UC) is a list of events and steps that shows the interaction between a user and the system. Use cases are generally written from the user's perspective [2] and turned out to be an excellent interaction instrument.

To avoid having to go through too much documentation, we have decided to use brief and descriptive use cases that also include more technical information than usual. This is primarily to decrease the risk of misunderstandings between the two teams.

The use cases are grouped by functional requirements (FR) defined in the project request. Because the goal of this thesis is first and foremost the development of concepts and prototype, not all use cases could be implemented within the scope of this work. Open issues are marked in the specific use cases.

### 4.1.1. Use Case Diagram

The listed Use Case Diagram below gives an overview over all use cases and shows the relations and interactions between them.

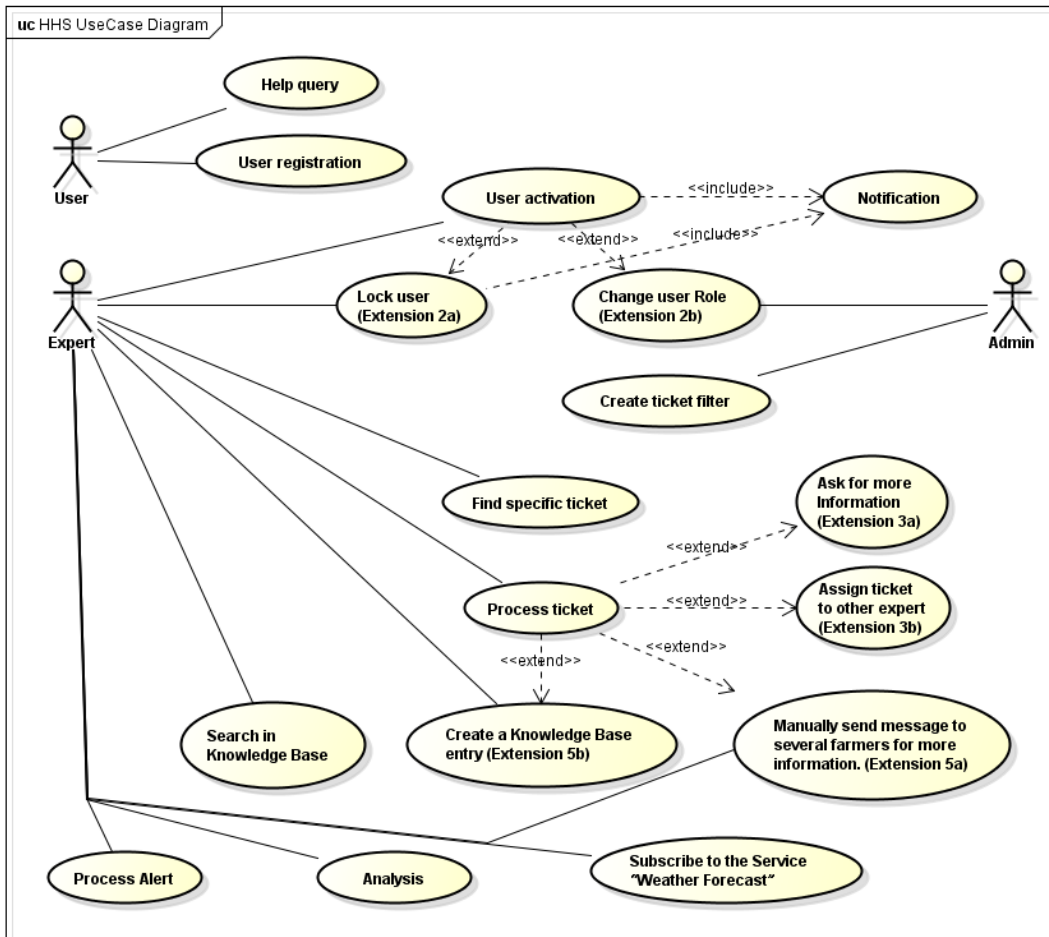


Figure 2: Use Case Diagram

On the following pages, each use case will be described in brief form.

#### 4.1.2. FR: Authentication & Authorization

HELVETAS' system requires manual activation of users. The main user group, the Lead Farmers in Nepal, can install the app and register. Activated users receive access to the application.

*For the sake of simplicity, actors within all of the following use cases will be addressed with male pronouns only.*

#### UC: User Registration

<b>Goal</b>	Registering a new User
<b>Primary Actor</b>	User
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>1. A user downloads the app from the Google Play Store or any other source (*.apk file) and installs it on his mobile device.</li> <li>2. The system asks the user to login or register.</li> <li>3. Because the user does not have an account yet, he chooses to register.</li> <li>4. The user enters all his personal information, such as name, mobile number, password, etc. He then presses the register button.</li> <li>5. A message pops up which explains that the account has to be activated before he can use the app. He then waits for activation.</li> </ol>	

**UC: User Activation**

<b>Goal</b>	Activate a User
<b>Primary Actor</b>	Admin / Expert
<b>Trigger</b>	UC: User Registration
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>1. An expert checks the system for new registrations every morning.</li> <li>2. He sees the new entry, checks whether the user should be allowed to access the system and then finally clicks on the activation button.</li> <li>3. The system sends an SMS to the user and informs that he may now log in.</li> </ol>	
<p>Extensions:</p> <p>2a. If the validation ended up negatively (i.e. user shall not be granted access):</p> <ol style="list-style-type: none"> <li>1. The expert clicks on the lock out button.</li> <li>2. The system sends an SMS to the user, telling him that he has been locked.</li> </ol> <p>2b. If the user requires Expert or Admin permissions:</p> <ol style="list-style-type: none"> <li>1. An Admin sets the relevant user role (either Expert or Admin).</li> </ol>	

**UC: Activation Message**

<b>Primary Actor</b>	Admin / Expert
<b>Trigger</b>	UC: User Registration
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>1. A few days later, the farmer receives an SMS informing him that he has been activated and may now start using the HELVETAS Horticulture System.</li> <li>2. The user starts the app, enters his phone number and password into the login form and starts using the app.</li> </ol>	

### 4.1.3. FR: Queries, Ticketing System

The app provides the functionality to issue queries to the system in case the user cannot find relevant information in the Knowledge Base to answer his questions. The queries are distributed to the Experts through a ticketing system. The system allows conversations between Users and Experts. In the scope of this system, conversations are called *Threads* whereas individual messages (i.e. questions and answers) are referred to as *posts*. If a user is offline, the query is stored on the mobile phone in local storage and will be sent to the system once Internet connectivity could be reestablished.

#### UC: Help Query

<b>Goal</b>	A farmer notices that his plants are covered in white fungi. He doesn't know what to do and issues a request for help with the app.
<b>Primary Actor</b>	User
<b>Open Issues</b>	Voice Messages are not implemented on the clients yet.
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>1. The farmer creates a Thread, enters his question and sends it to the system.</li> <li>2. To make the problem clearer, he attaches a picture of the fungi. <i>Implication: The farmer cannot choose a preferred expert as he may not have expert knowledge on that particular topic.</i></li> <li>3. As the user sends the question, the system shows a sign indicating that the post has not yet been synchronized.</li> </ol>	

#### UC: Find Specific Ticket

An expert wants to find an old ticket that he processed a month ago. He opens the search dialog, enters a query string and defines the time period. The system returns the search result.

*Open Issues:* This feature has not been implemented yet. The Administration Client currently queries the first 100 threads only but allows client side filtering by thread title and post text.

**UC: Process ticket**

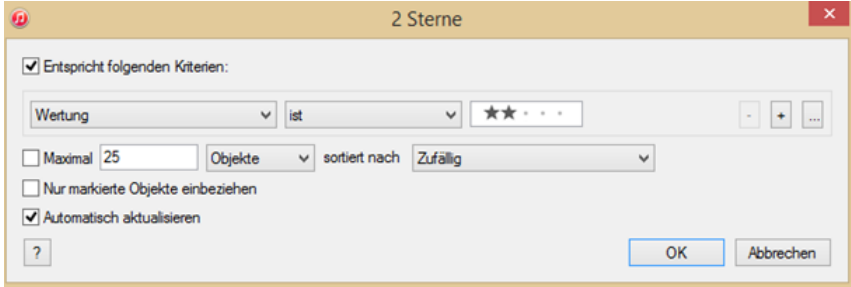
<b>Goal</b>	Answer Help Queries.
<b>Primary Actor</b>	Expert
<b>Trigger</b>	Help Query
<b>Open Issues</b>	4) Locking is currently only supported on the server. Extension 3b) Assigning is possible on the server, though not yet on the Administration Client.
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>1. An expert opens the ticket system. He clicks on the view “nearest farmers” <i>See UC: Create Ticket Filter for information on how to specify a view.</i></li> <li>2. The system shows him threads with distance less than 100km. The result set is sorted by the proximity.</li> <li>3. The expert reads the first query and locks the ticket <i>Other experts cannot edit the thread while it is locked. Only admins are can unlock threads currently assigned to other experts.</i></li> <li>4. The expert types an answer and clicks on send button.</li> <li>5. The system sends the answer to the farmer.</li> <li>6. The farmer can mark the thread as closed, thus hiding it from the view.</li> </ol>	
<p>Extensions:</p> <ol style="list-style-type: none"> <li>3a. The expert has further questions regarding the query. <ol style="list-style-type: none"> <li>1. He sends a response to the farmer. He asks for more information and tells him that attaching a picture might help.</li> </ol> </li> <li>3b. The expert reads the first query. He realizes that his colleague might be more qualified to answer the question. <ol style="list-style-type: none"> <li>1. He assigns the ticket to his colleague.</li> <li>2. The system will create a lock on that ticket for the new expert. Only the assigned expert will be able to work on the ticket.</li> <li>3. In case the new expert does not reply to a post in time, the ticket will eventually be unlocked again and put into the global queue. <i>Only admins are allowed to unlock threads that are assigned to other experts</i></li> </ol> </li> <li>5a. The expert decides to send this thread to multiple farmers. <ol style="list-style-type: none"> <li>1. <i>See UC: Manually send message to several farmers for more information.</i></li> </ol> </li> <li>5b. The experts decides to share this questions in the FAQ. <ol style="list-style-type: none"> <li>1. <i>See UC: Create a Knowledge Base Entry.</i></li> <li>2. The farmer gets a link.</li> </ol> </li> </ol>	



**UC: Create Ticket Filter**

<b>Goal</b>	Creating a new view in the ticketing system that filters threads in a pre-defined manner.
<b>Primary Actor</b>	Admin
<b>Open Issues</b>	Ticket filters are implemented dynamically on the server but statically in the Administration Client. Changing filters requires the source code to be modified.

Main Success Scenario:

1. An admin opens the system and clicks on “Create new Ticket Filter”.
2. He defines some attributes (mockup image below):  

3. After saving the filter, a new tab with the filtered view appears.  
*Technical Info: Experts see the new filter after restarting the application.*

#### 4.1.4. FR: Knowledge Base

Farmers can search for documents that are grouped by topics and keywords. A useful and meaningful structure of the Knowledge Base is part of this project. The Knowledge Base is administered and fed with content by the experts in Nepal.

##### UC: Create a Knowledge Base Entry

<b>Goal</b>	An expert wants to write a Case Study, Best Practices, How-To Guide, or an FAQ entry.
<b>Primary Actor</b>	Expert
<b>Trigger</b>	UC: Process Ticket / Extension 5b
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>1. The expert navigates to Knowledge Base → Documents → Add Document.</li> <li>2. He chooses the metadata: language, crop, season, title.</li> <li>3. He then uploads the FAQ as a Microsoft Word document.</li> <li>4. The FAQ is now available in the Knowledge Base.</li> </ol>	

##### UC: Search in Knowledge Base

<b>Goal</b>	A farmer has a problem with fungi. Before he sends a query, he decides to search the Knowledge Base for an existing FAQ entry regarding his problem.
<b>Primary Actor</b>	User / Expert
<b>Open Issues</b>	Not implemented yet on the Android Client.
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>1. The farmer goes to the Knowledge Base search area.</li> <li>2. He enters search attributes (crops, season) and specifies a language.</li> <li>3. The system shows him a list of all search results.</li> <li>4. The farmer opens the FAQ and finds a tip for his problem. He does not need to send a query to an expert.</li> </ol>	

#### 4.1.5. FR: Services

Farmers can be subscribed to several different services at the same time. The system sends push notifications to the subscribers. One possible service would be a service to receive weather forecasts for the farmer's location. This bachelor thesis will realize two sample services (Weather Forecast and Marked Prices).

#### UC: Subscribe to the Service “Weather Forecast”

<b>Goal</b>	A farmer would like to receive a weather forecast for the next week for his location, including chances of rain, temperature at day / night, humidity, wind, . . .
<b>Primary Actor</b>	User
Main Success Scenario:	
<ol style="list-style-type: none"> <li>1. The farmer navigates to services.</li> <li>2. The system shows a list with all available services.</li> <li>3. The farmer selects all desired services and clicks OK.</li> <li>4. From now on, the system frequently sends news associated with the subscribed services to the farmer's news feed.</li> </ol>	

#### UC: Manually Send Message to Several Farmers

<b>Goal</b>	An expert decides to send a thread to multiple farmers for information. <i>That means, they are not able to respond directly to this message. A thread is still a conversation between n-experts and a single farmer.</i>
<b>Primary Actor</b>	Expert
<b>Trigger</b>	UC: Process Ticket / Extension 5a.
<b>Open Issues</b>	Not implemented.
Main Success Scenario:	
<ol style="list-style-type: none"> <li>1. The expert is viewing a specific thread.</li> <li>2. He clicks on the button “Send to several farmers”.</li> <li>3. Now he enters the farmers' phone numbers in the recipient list and clicks on “Send”.</li> <li>4. The system shows the message in the news feeds of all farmers in the list. <i>This functionality could be implemented as service where every farmer is auto-subscribed.</i></li> </ol>	

#### 4.1.6. FR: Alerts

Admins can trigger special events. The system then sends SMS alerts to the selected user group (e.g. all farmers within 100km of Kathmandu). Examples for an alert are earthquake notifications and storm warnings.

*Open Issues:* This requirement is not addressed by this thesis. The server can issue SMS notifications but user selection is yet to be implemented.

#### UC: Process Alert

<b>Goal</b>	An admin realized that there is a storm incoming. He decides to issue an alert to all farmers in the region.
<b>Primary Actor</b>	Admin
<b>Trigger</b>	UC: Process Ticket
<b>Open Issues</b>	Not implemented.
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>The administrator navigates to the Alert section. <i>We have decided to assign the permissions to send alerts to admins only so that the risk of abuse can be decreased.</i></li> <li>He defines the target group: all farmers within 100km to Kathmandu.</li> <li>He enters an alert message.</li> <li>The system sends an SMS alert to all farmers that meet the criteria.</li> </ol>	

#### 4.1.7. FR: Analysis - Data Mining

The farmers provide useful (crowdsourced) data that can be analyzed. Even though actual data analysis is not part of this project, a simple framework for further SQL based analysis shall be provided. Within the scope of this bachelor thesis, one example report will be implemented. Such a report might answer the question “Which problems appear often in a certain region?”.

*Open Issues:* This requirement could not be addressed in thesis.

**UC: Analysis**

<b>Goal</b>	An admin wants to find the regions where a specific type of fungi appeared more often than usual.
<b>Primary Actor</b>	Admin
<b>Open Issues</b>	Not implemented.
<p>Main Success Scenario:</p> <ol style="list-style-type: none"> <li>1. The admin opens the analysis area.</li> <li>2. The system shows a text area and a button labeled “Query to Database”.</li> <li>3. The admin types an SQL query, such as: <pre>SELECT * FROM Post p JOIN [User] u ON p.UserId = u.Id WHERE p.text LIKE '%funghi%' AND p.language = 'en-us' GROUP BY u.coordinate @@ geo_within('coordinate_kathmandu', 10km)</pre> </li> <li>4. After executing the query, the system will display the SQL result set.</li> </ol>	

**4.2. Nonfunctional Requirements**

The following sections state the analysis of the nonfunctional requirements.

**4.2.1. Technologies**

Technologies to be used include a native Android app, a Administration Client based on Microsoft Windows that comes with a self-extracting archive, and a server back end based on Windows Server and MSSQL. The testing environment for the server is based on Windows Server 2008 R2 and Internet Information Service (IIS) 6.5. For the database, MS SQL Express 2012 is used.

**4.2.2. Open Source**

The system is developed as open source software. Everyone will thus be allowed to download, install, modify and repurpose system components freely. The entire project will be released under the GPLv3 license.

*As the system's source code is public and can be accessed by anyone, we have taken precautions so that no credentials or API keys were committed to Github.*

### 4.2.3. Data Rates & Data Volume

As system is primarily going to be rolled out in developing countries, it is to be expected that mobile networks are often slow, unreliable and expensive. The application should consider these limitations. Easy solutions with low data usage should be preferred whenever possible.

In coordination with HELVETAS Nepal, low usage of data rates and volumes is primarily required for the Android client. For the Administration Client, we can expect good LAN or WLAN connections.

Nevertheless, we have tried to keep bandwidth requirements as low as possible by following the approaches below:

**Google Cloud Messaging (GCM):** Allows sending push notifications to mobile phones. For this purpose, GCM keeps a persistent TCP connection open between phones and the cloud. With this approach, all apps on a device can share a single connection that remains dormant for the majority of the time, thereby data and battery usage is greatly reduced. GCM helps us avoiding frequent polling.

**RESTful API:** We have created a RESTful server API that allows flexible interfaces to be used in combination with small data packages.

**Data Hiding:** High data usage may not be an issue on the Android Client but definitely is on the Android client. In combination with lower permissions for phone users, a special attribute [JsonMapWithPermission] has been created (see permission paragraph in Section 5.4.2). The attribute allows the dynamic mapping of properties that can only be accessed with higher permission levels. A positive side-effect is lower data usage.

**Android Synchronisation:** When an Android client manages to reestablish connection to the Internet, it tries to synchronize with the server automatically.

In order to avoid the overhead of individual requests, we have defined bulk-upload routes. One route allows the creation of multiple new threads (`api/threads`) while another route (`api/thread/posts`) is used to create multiple new posts.

The same concepts apply to downloading data. Both server and client ensure that a thread or a post is never sent more than once. This is implemented using the API's `createdSince` date parameter.

The time difference between server and client must be managed correctly as inaccurate clocks pose a risk of data loss. To avoid this, the client also sends the current

device time (`clientTime`) to the server. The server is then able to calculate a time delta which can be used to accurately determine what posts have to be issued since the last synchronization.

**Image Compression:** Large multimedia files could rapidly increase data volumes. We have therefore decided to use JPEG compression for photos. After spending time on research[1] and running several tests, we have concluded that a JPEG compression rate of 50% with a maximum image size of 1024x1024 pixels is sufficient. The resulting visual quality is satisfying enough and the reduction of data volume is immense. The image file sizes generally fall between 80kB and 120kB and the maximum file size has been set to 200kB for photo posts.

*Note: Multimedia content is stored as Base64 encoded `nvarchar` data as Base64 is used to transfer data using a JSON based protocol.*

#### 4.2.4. Usability

Future users of the system are usually not familiar with technical gadgets. Therefore, the user interface should be easy-to-use and provide intuitive handling to make it easier understandable for the users.

This has mainly been the focus on the Android client. The Administration Client currently provides rudimentary usability (e.g. no short cuts and basic layouting). However, the foundations for improvements have been laid as the client is consistently following the MVVM-Pattern using data binding techniques.

Much more effort has been put into usability concepts for the Android client:

- We strictly abode principal Android Design Guidelines<sup>3</sup> and tried to create a user experience that is as easy to follow as possible.
- We expect that Lead Farmers will be taught how to use the app and think that the basics can be understood quickly.
- In addition, we have followed commonly accepted standards for similar chat-like applications such as WhatsApp or Skype in order to avoid alienating users.
- General Usability Patterns<sup>4</sup> that we have followed include *Master/Detail*, *Empty List*, *Auto Refresh*, *Data Formats*, *Navigation Stack*, etc.

---

<sup>3</sup><http://developer.android.com/design/index.html>

<sup>4</sup><https://www.google.com/design/spec/material-design/>

### 4.2.5. User Experience

Due to time constraints, only informal usability tests could be performed. The HELVETAS developers that will take over the project will start conducting usability tests on their own with actual Lead Farmers.

The following screenshots depict the user interface of the Android app.

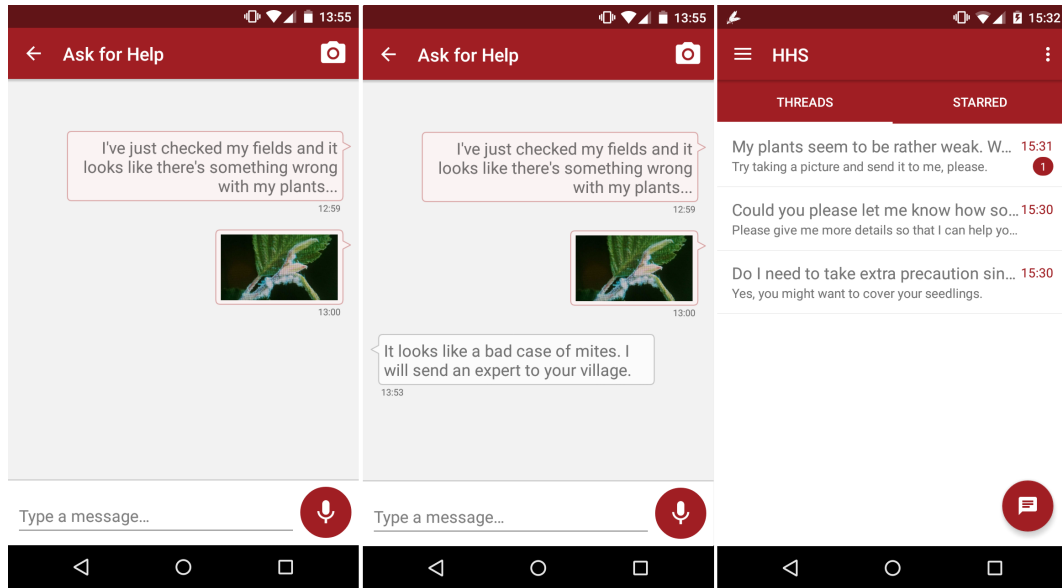


Figure 3: UX: Posts & Threads

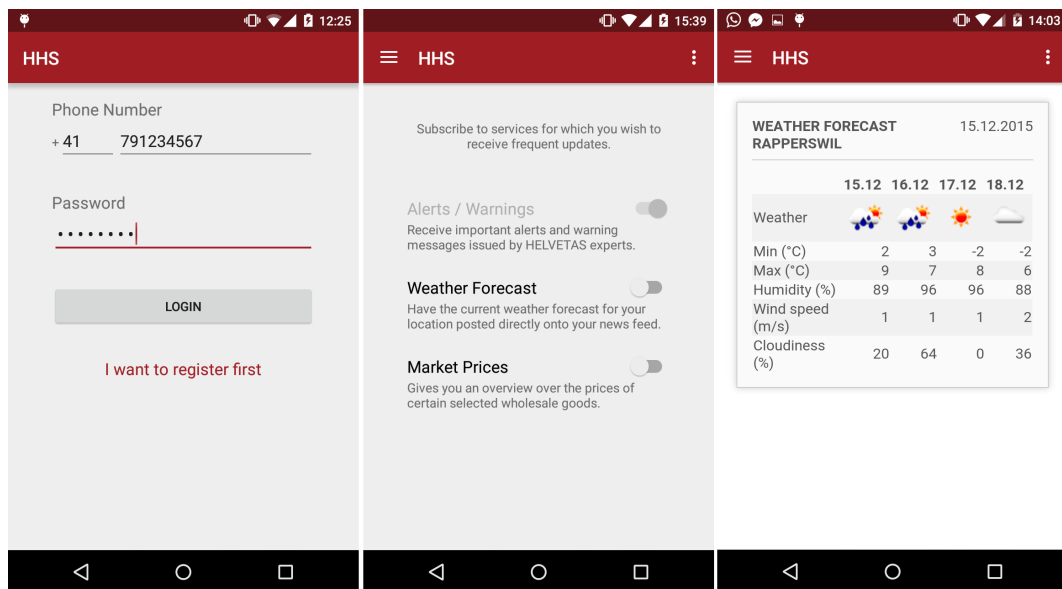


Figure 4: UX: Login, Services and News Feed



#### 4.2.6. Scalability

The system will be used by more than 10'000 users and a maximal number of 30'000 help requests per month has been estimated by HELVETAS.

**Result Set Limitation:** Queries to the database are currently limited to yield at most 100 rows as pagination has not yet been implemented. This has been done to avoid system crashes resulting from high memory usage (`OutOfMemoryException`) and timeouts due to high data loads. We were pursuing the approach of direct server searches for users, threads and Knowledge Base documents. The server search is currently only implemented for the Knowledge Base.

**Hierarchical Agglomerative Clustering (HAC):** For the implementation of the weather forecast (see Section 6.4.1), we also had to take the scalability into consideration. See Section 6.4.1 for detailed information.

**Performance Tests:** To test a server load as described in the requirements, we first tested data insertion via the RESTful API on a large scale and measured the response time for each request (see test program appendix: *hhsperf*).

We have conducted our performance tests with a small dataset as well as with a rather large database that resembles about a year's work with the HELVETAS Horticulture System (tested on LAN):

	Small Dataset	Large Dataset
Database Size	25MB	800MB
User Count	30	10'000
Thread Count	100	50'000
Post Count	450	450'000
Average Request Response Time for Insertion of Users, Threads and Posts	50ms	50ms
Spatial Calculations (Proximity Queue)	220ms – 260ms	1.7 – 2.0s

As shown above, the processing time for requests and queries does not noticeably increase with larger amounts of data, i.e. the numbers of entries tested are absolutely manageable by the system. The notable exception are spatial queries that seem to grow exponentially, so there is definitely potential for optimization.

The tests have also been conducted over Wi-Fi. While the average response time increased by a factor of 6, it stayed the same regardless of the amount of data in the database. We conclude that the system is fit to handle the estimated load.

**Integration Tests:** The following definition<sup>5)</sup> of Unit Tests and Integration Tests provided on Stack Overflow by a user by the name of *ddaa* is spot-on in our opinion:

Unit test: Specify and test one point of the contract of single method of a class. This should have a very narrow and well defined scope. Complex dependencies and interactions to the outside world are stubbed or mocked.

Integration test: Test the correct inter-operation of multiple subsystems. There is whole spectrum there, from testing integration between two classes, to testing integration with the production environment.

Conducting unit testing is essentially testing methods in isolation from other system components, which makes this approach particularly useful for testing the correctness of calculations. The `HacAlgorithm` can and is tested with a unit test. However, in the server component of the HELVETAS Horticulture System, nearly all methods depend on the database. Having experimented with `Moq`<sup>6</sup> (Mocking Framework), it became obvious that unit testing is probably not the best choice for HHS.

We have thus decided to stick with integration tests in combination with a real database setup. In addition to the benefit of not having to spend time setting up mocks, we are also able to test against a more realistic scenario.

The table below shows the overall test coverage:

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
▲ {} HHS.BLL.Services	234	11,83 %	1744	88,17 %
▶ 🐞 DocumentService	10	3,32 %	291	96,68 %
▶ 🐞 MappedGenericService<TDT...	15	46,88 %	17	53,13 %
▶ 🐞 NewsFeedService	0	0,00 %	108	100,00 %
▶ 🐞 NotificationService	106	86,18 %	17	13,82 %
▶ 🐞 TagService	26	13,83 %	162	86,17 %
▶ 🐞 ThreadService	32	3,63 %	850	96,37 %
▶ 🐞 UserService	11	4,26 %	247	95,74 %
▲ {} HHS.BLL.Services.NewsFeeds	719	79,80 %	182	20,20 %
▶ 🐞 HacAlgorithm	31	14,69 %	180	85,31 %
▶ 🐞 HacAlgorithm.Cluster	0	0,00 %	2	100,00 %
▶ 🐞 MarketPriceJob	307	100,00 %	0	0,00 %
▶ 🐞 NewsFeedJobScheduler	89	100,00 %	0	0,00 %
▶ 🐞 WeatherJob	292	100,00 %	0	0,00 %
▶ {} HHS.BLL.Services.WebServices.D...	6	100,00 %	0	0,00 %

Figure 5: Test Coverage

We have managed to reach about 95% test coverage for the services. As every API call runs through a service, we can assume that around 95% of the server is covered

<sup>5</sup><http://stackoverflow.com/questions/520064/what-is-unit-test-integration-test-smoke-test-regression-test> (February 6<sup>th</sup> 2009)

<sup>6</sup><https://github.com/Moq/moq4>

with unit tests. The value of 88.17% coverage for the HHS.BLL.Services can be explained by the low coverage of the notification service. This service is responsible for notifying the clients over Google Push Notifications or via SMS messages. This external dependency makes static testing rather difficult and only provides meaningful test results in combination with an actual phone. The same applies to the News Feed jobs as they are depending on web services and also only allow dynamic system tests.

Other now fully covered components such as the MappedGenericService or the GenericRepository (which are not shown in the diagram) are very powerful. At the moment, not the entire range of functionality provided is used by the system. We have decided to keep the code in the system as we think it will be useful in the near future.

#### 4.2.7. Internationalization

The HELVETAS Horticulture System will be able to support multiple languages. English and Nepali will be implemented in the prototype of this bachelor thesis. The system has to support changes and extensions.

**Admin:** We have decided to use the integrated WinForms localization feature and thus do not need to reinvent the wheel. As the original WinForms localization is bound to each single form, we have decided to use a stub-form that serves as a localization hub for the other forms as well and is not actually visible to the user. Our solution of only using the integrated Resource Manager<sup>7</sup> has several advantages:

- It is easier to manage and translate only one single localization file.
- The standard localization process is only useful if the translation can directly be set during design time. However, most strings are dynamic and will be set at runtime depending on the current state of the form (e.g. strings in a DataGridView). This implies that most translation must be set manually in code regardless.

**Server:** The localization requirement has a substantial impact on the Generic System requirement. Dynamic contents, like filters or tags, have to be translated, which makes using them more complex.

**Android:** Localization on Android<sup>8</sup> is straightforward via specialized XML files.

---

<sup>7</sup>[https://msdn.microsoft.com/en-us/library/y99d1cd3\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/y99d1cd3(v=vs.71).aspx)

<sup>8</sup><http://developer.android.com/guide/topics/resources/localization.html>

#### 4.2.8. Generic System

The system is primarily designed with horticulture in mind. However, the system tries to offer generic functionality whenever possible. This allows components to be reused in applications with different purposes, such as health care. The design should consider this from the start.

We took the Generic System into account in the Knowledge Base structure with Tags (Section 6.3.2) and also with the Thread Filters (Section 6.1.1). For more information, please consult the referenced sections.

#### 4.2.9. Documentation

The development of the prototype will continue after the duration of this bachelor thesis. This means that the documentation is of great importance.

Because HHS will continue to be developed by HELVETAS after this thesis, high quality of code. The first step to easy-to-understand code is good code documentation. In HELVETAS Horticulture System, every class, public and protected method is documented. In addition, continuous refactoring has lead to a reduction of code complexity.

The code quality is usually measured in code metrics, as shown below.

Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
[All]	[Member]	[Type]	[Type]	[Member]
100-20 Maintainability 1774	1-10 Complexity 1516	1-2 Inh. Depth 161	0-9 Coupling 115	1 - 10 Lines 1391
19-10 Maintainability 2	11-20 Complexity 13	3-4 Inh. Depth 19	10-80 Coupling 77	11-20 Lines 76
9-0 Maintainability 0	21+ Complexity 1	5+ Inh. Depth 16	81+ Coupling 1	21+ Lines 63

Figure 6: Code Metrics

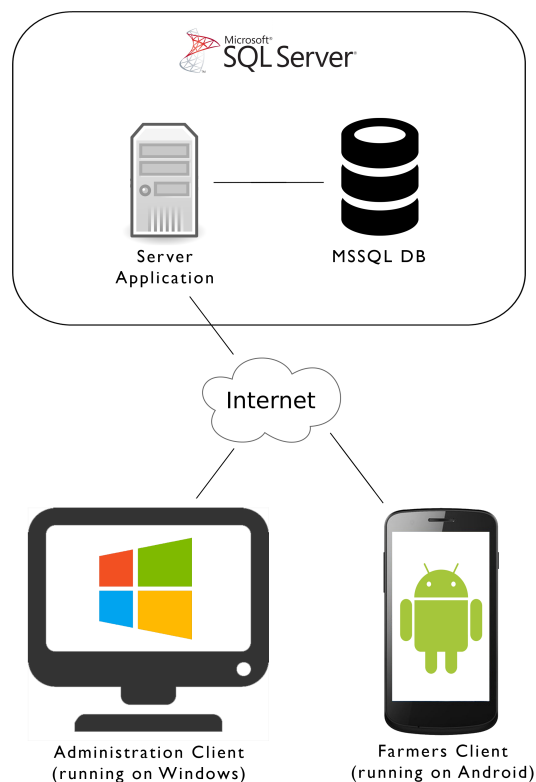
More information on each metric and the ratings can be found on Code Metrics Viewer<sup>9</sup>. The summarizing Excel sheet can be found in in the appendix /misc\*.

<sup>9</sup><https://codemetricsviewer.wordpress.com/2011/06/26/how-to-interpret-received-results/>

## 5. Architecture

This section elaborates technical principles, structure, concepts and frameworks that have been used for the implementation of the HELVETAS Horticulture System.

### 5.1. System Overview



*Figure 7: System Overview*

The diagram shows the three applications of the HELVETAS Horticulture System.

**Server:** The server is an ASP.NET Web API application. It is hosted on an Internet Information Service (IIS) Web Server on Windows Server. A Microsoft SQL database server is used for persistence.

**Windows:** The Windows Administration Client is used by experts and admins. They process the tickets, create Knowledge Base entries, or send alerts to the users.

**Android:** The Android Client is mainly used by farmers. They ask for help, search documents in the Knowledge Base, or receive weather forecasts and other notifications on their News Feed.

Both, the Android Client and the Administration Client, communicate via the server with each other. A complete overview of all system components can be found in the deployment diagram (Section 5.2).

## 5.2. Deployment Diagram

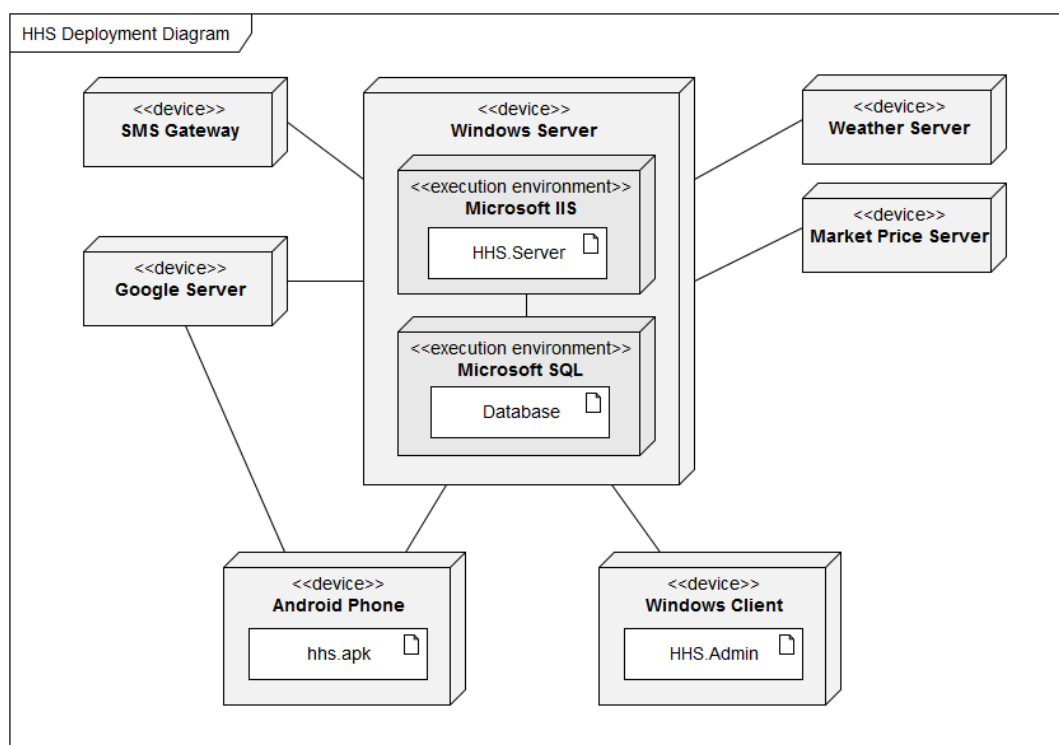


Figure 8: Deployment Diagram

In addition to the system overview (Section 5.1), these components are of interest:

**SMS Gateway:** The HHS uses an SMS gateway for urgent notifications. Because users may be offline (see Offline Mode in Section 5.7.6), a second communication channel has been chosen by HELVETAS. For example, alerts (Section 4.1.6), account activation, and lock state changes are sent as SMS messages.

**Google Server:** This project uses several Google Services, namely Google Cloud Messaging for push notifications and Google Location, Google Geocode and Google Elevation for location information.

**Weather Forecast Server & Market Price Server:** These third-party services provide data for the News Feed of subscribed users (see Section 6.4).

### 5.3. Architectural Decisions

#### 5.3.1. ASP.NET

We have decided using ASP.NET instead of a Java or PHP solution because of following reasons:

- HELVETAS Nepal owns Windows server. Using .NET technologies is the natural choice considering all out-of-the-box support.
- We both are more experienced with .NET, also on ASP.NET. That means, we could save a lot of time on the .NET part of the project.
- ASP.NET is widely used and popular. It is a sophisticated technology that is compatible to Internet Information Services (IIS) which is integrated in Windows Server. There are also many code examples for illustration purposes available.

#### 5.3.2. Reasons for Building Native Applications

The HELVETAS Horticulture System consists of two clients, where the Administration Client runs on Microsoft Windows and the mobile application runs on Android.

At the very beginning of this project, we have decided to focus our analytical efforts regarding the architecture of the clients on two distinct approaches: native or web based. We soon have realized that both approaches have significant benefits but also major downsides.

The following table lists the criteria including their weighting upon which we have based our decision. The two approaches will be compared with each other according to the criteria in a decision matrix.

Criteria	Wt.	Why is the criteria important?
Scalability	2	Scalability is one of the major requirements, as this project is intended to undergo further development.

Compatibility	1	We anticipate that client and app are going to be run on cheap and out-dated hardware, thus compatibility to older devices and legacy versions of Android OS must be taken into account.
Maintainability	2	HELVETAS is intending to continue development, thus high maintainability is critical.
Availability	1	Availability of the system is crucial. Especially for the mobile app, users are not expected to have continuous Internet access, so they must be able to continue using the app even if they are offline.
Satisfies Use Cases	3	It is important that project use cases can be satisfied by the technologies in question.
Satisfies Constraints	2	Several constraints and restrictions shall be taken into consideration separately in order to increase their weighting in this decision making process. The constraints include unreliable Internet connectivity, out-dated hardware, deployment costs, etc.

The criteria described above are now compared against each other in the following decision matrix, along with a short description for each criteria.

Decision Matrix	Wt.	Native	Web Based
Scalability	2	(1.0) High, device capabilities can be fully used.	(1.0) High, HTML5 is designed with scalability in mind.
Compatibility	1	(1.0) High, Windows and Android both offer great support for legacy versions.	(0.25) Low, outdated browsers are installed.
Maintainability	2	(1.0) High, Java / C# help with clear type systems.	(0.5) Medium, clear structures are more difficult to achieve.
Availability	1	(1.0) High, clients can be run without Internet connectivity.	(0.25) Low, clients require continuous Internet connectivity.
Satisfies Use Cases	3	(1.0) High, all device features can be accessed.	(0.25) Low, browser functionality is limited.



Satisfies Constraints	2	(1.0) High, native applications can be fully adjusted to meet all constraints to the best possible degree.	(0.25) Low, web based applications require consistent access to the Internet, updated software, etc. However, the deployment is easier as browsers are readily available.
<b>Total</b>		<b><u>11.00 pt</u></b>	<b><u>4.75 pt</u></b>

As shown by the decision matrix above, the benefits of native applications clearly outweigh the advantages of web based application. Therefore, we have concluded that implementing native prototypes is a better choice within the scope of this particular project.

Initially, we have also discussed the possibility of creating hybrid applications, i.e. a combination between native applications and web based solutions. However, we have decided against this approach as this would basically combine the worst of both approaches while only giving marginal benefits over any one of them.

Another factor that has been part of the decision making process was *previous personal experience*. We both have previous development experience with the .NET Framework and Android. In addition, HELVETAS Nepal has previously been working on Android Projects as well and therefore has existing know-how that can be used for the future development.

While this was fact was by no means the tipping point of the decision, we do believe that available know-how should be used whenever possible as it is of best interest for the project.

The decision of developing native application has been agreed upon with HELVETAS Nepal and was found to be the most suitable approach.

### 5.3.3. Communication to the Client

One of the main non-functional requirements is the one regarding low data volume. To take this into account, we have decided to exchange data in JSON format instead of XML to decrease unnecessary overhead.

While the Android clients have expensive, slow and sometimes unreliable networks, the Windows Administration Clients have WLAN. That means better data rates and higher data volumes. This fact makes it worth the time of thinking about using WCF for easier communication between the Windows Administration Client and the Server. Pros and cons of this decision are listed in tabular form:

WCF	ASP.NET Web API
<p>Pros:</p> <ul style="list-style-type: none"> <li>• Abstracted network layer</li> <li>• Once set up greatly reduces development time</li> <li>• Extensive configuration possibilities</li> </ul>	<p>Pros:</p> <ul style="list-style-type: none"> <li>• Comfortable binding of requests</li> <li>• Low overhead</li> <li>• Reuseable on Android</li> <li>• Extensive configuration possibilities</li> </ul>
<p>Cons:</p> <ul style="list-style-type: none"> <li>• High complexity, long training time</li> <li>• High synchronization cost, high overhead (SOAP)</li> <li>• Lots of configuration possibilities. Risk of bad configuration[6].</li> </ul>	<p>Cons:</p> <ul style="list-style-type: none"> <li>• High complexity, long training time</li> <li>• More effort for communication</li> <li>• Many configuration possibilities. Risk of bad configuration, but we do have a reference system.</li> <li>• Non automatic synchronization, though this is not essential in this application.</li> </ul>

The choice eventually fell onto ASP.NET Web API as it seemed to be more adequate for this project and because some previous knowledge was available. As we have decided for ASP.NET, previous thoughts and concepts about custom protocols could be dismissed

#### 5.3.4. Server Layers: Why Two Layers & Why no DAL?

As it is taught in *Application Architecture* at HSR, a service system is generally divided into three layers: *Data Access Layer (DAL)*, *Business Logic Layer (BLL)*, and *Presentation Layer (PL)*.

However, we have decided to work with *Entity Framework (EF)*, a highly evolved OR-Mapper. In our opinion, EF already implements the unit of work (DbContext.cs)

and the repository pattern (DbSet) itself. We believe an additional abstraction layer is superfluous and agree with the Microsoft Tutorial on the topic[3]. We see the Entity Framework as our DAL.

### 5.3.5. Why Windows Forms was Chosen instead of WPF

The primary reason we have decided for Windows Forms is due to higher backwards compatibility and the smaller overhead compared to WPF, as HHS application is very likely to be run on outdated hardware. Another large factor is previous knowledge not only from our side but also from the HELVETAS developers.

Reasons that spoke against Windows Forms were inferior data binding capabilities and the “less declarative, more imperative” approach which reduces maintainability.

## 5.4. Server

### 5.4.1. Presentation Layer

**Responsibilities:** The PL handles the entire network communication, validates the input and provides the authentication functionality. The mapping between the data model and JSON is done automatically by ASP.NET.

**HTTP Filters:** In ASP.NET, global filters can be configured that will be applied to each request (see `WebApiConfig.cs`). The system uses such filters are used for authentication and input validation. Another filter has been specifically implemented so that the API can only be accessed via secure HTTPS connections. Filters are also being used for error handling and dispatching on the server and cause exceptions to be logged correctly into the log table of the database.

**RESTful Service:** The API that allows access to the system’s functionality and drives the communication between server and clients is following representational state transfer principles (REST).

The web service endpoint respects REST-Level II. The resource naming conventions are clear and precise. Resources are manipulated using the correct HTTP verbs, such as GET and POST, in their originally intended meaning.

**Authentication:** Websites and web services often require users to be authenticated first. This is usually done via a randomly generated security token that is sent from the browser to the end point. This, however, introduces state information into the communication between server and client. For this project, it has been

agreed upon with HELVETAS that clients identify themselves using HTTP BASIC authentication over secure SSL connections (HTTPS).

**Input Validation:** One of the responsibilities of the presentation layer is input validation. For this task, Data Annotations offered by the .NET Framework have been used with ASP.NET. Correctness of incoming data can be verified by calling `ModelState.IsValid()`.

#### 5.4.2. Business Logic Layer

**Responsibilities:** The BLL defines the mapping between model and database and contains the business logic. LINQ to Entities<sup>10</sup> is used for data retrieval. Furthermore, the mapping between data models and data transfer objects (DTOs) under proper consideration of authorization are also part of the BLL.

**Permissions:** The system defines three user groups with different permission levels: *users*, *experts* and *admins*. More information on the different groups can be found in Section 3.2 (Domain Terminology).

The user groups are static and can only be changed on source code level. This was suggested by HELVETAS as more permission levels were not deemed to be necessary.

Permission levels are stacked, meaning that Experts are also able to do anything Users can do, and in turn, Admins can do anything Experts are allowed to do.

In some cases, the content of the response body for a request depends on whether it has been issued by a User or an Expert. For example, after querying a thread, a User must not receive information regarding thread status, last known location or thread priority. In order to differentiate between these different cases, we have implemented an extension for the JSON serializer. Essentially, a custom attribute, e.g. `[JsonMapWithPermission(UserRole.Admin)]`, can be specified for any field that requires special permissions. If the user requesting the resource does not have the necessary minimum authorization level to access the data, nothing will be returned.

**AutoMapper:** The model to DTO mapping is done with the help of a useful tool called *AutoMapper*. Regardless of all advantages, there are a number of things that need to be considered. One should be careful when mapping complex data transfer objects (DTOs) to database entities. Each simple object should be stored to the database on its own as Entity Framework does not store the entire object graph.

---

<sup>10</sup><https://msdn.microsoft.com/library/bb386964>

**EF: Code-First Approach:** The graphic below shows the code first approach where SQL installation scripts are things of the past. First, the Domain Model must be created and the mapping must be configured that is then followed by OR mapper magic[4].

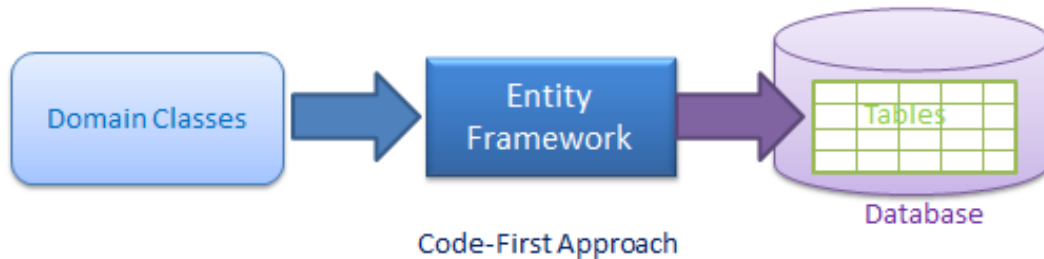


Figure 9: Code-First Approach <sup>11</sup>

Entity Framework has an integrated protection against bad DB schemas. It allows only associations between entities that are in third normal form. With this measure in place, exotic associations are suppressed. In Entity Framework, the configuration part can be used with DataAnnotations or with FluentAPI. We decided using DataAnnotations because of the better clarity. Using DataAnnotations unfortunately also has some limitations. For example, default values or foreign key cascade options are not available. The latter is a problem because MS SQL has a painful restriction of defining multiple cascades to a table (their reason is trying to avoid loops). In combination with Entity Framework Migration it is possible to manually change the cascades once.

**EF Migration:** Different database versions are a big challenge while developing. Because of this, we decided, to deploy (einsetzen) with the Entity Framework Migration tool. It allows to manually save a new db version by command `Add-Migration [MyMigrationName]`. To apply this version to the database the command “Update-Database” can be executed. You probably want to apply a specific version or rollback the version with `Update-Database -TargetMigration [MyMigrationName]`.

To help future developers save a lot of time and trouble, there are some useful informations to deal with EF Migrations:

- struct is NOT NULL / [Required] by default. To change this, make struct nullable, e.g. `Guid?` or `Nullable<Guid>`.
- Foreign keys which are NOT NULL / [Required] are set to `cascadeDelete = true` by convention. Make type Nullable or remove the cascade from the migration.

<sup>11</sup><http://www.entityframeworktutorial.net/>

A useful migration commands:

- `update-database -script` gets an SQL script to update the database manually.

We strongly recommend to adding a new migration for each model change. That means, do not use the `Add-Migration [previousMigrationName] -force`. Otherwise, you could get errors because of DB incompatibilities.

**Error Concept:** We decided throwing exceptions instead of boolean return values, because they guarantee a higher error handling quote. Booleans can be ignored; sometimes, they are overlooked. The server knows three error status codes:

- **400 Bad Request** means that the error originated on client side as the request was invalid (e.g. malformed, invalid data, DB constraints violated, etc.).
- **401 Unauthorized** errors address all kind of permission problems. The credentials might be wrong, the user could be locked out or does not have the required permission level to access the data.
- **500 Internal Server Errors** should never occur. In case an unexpected exception occurred anyway, the status code 500 is thrown.

In all these cases, an `HSHttpError` is sent back which provides an error message and an error code. The error message is meant for developers and can be used for log files, the error code is translated in the localization file and is used for translated user messages.

### 5.4.3. Dependencies

The dependency diagram is simplified for a better understanding and only shows the dependencies between the important components. References to utilities and helper classes have been removed.

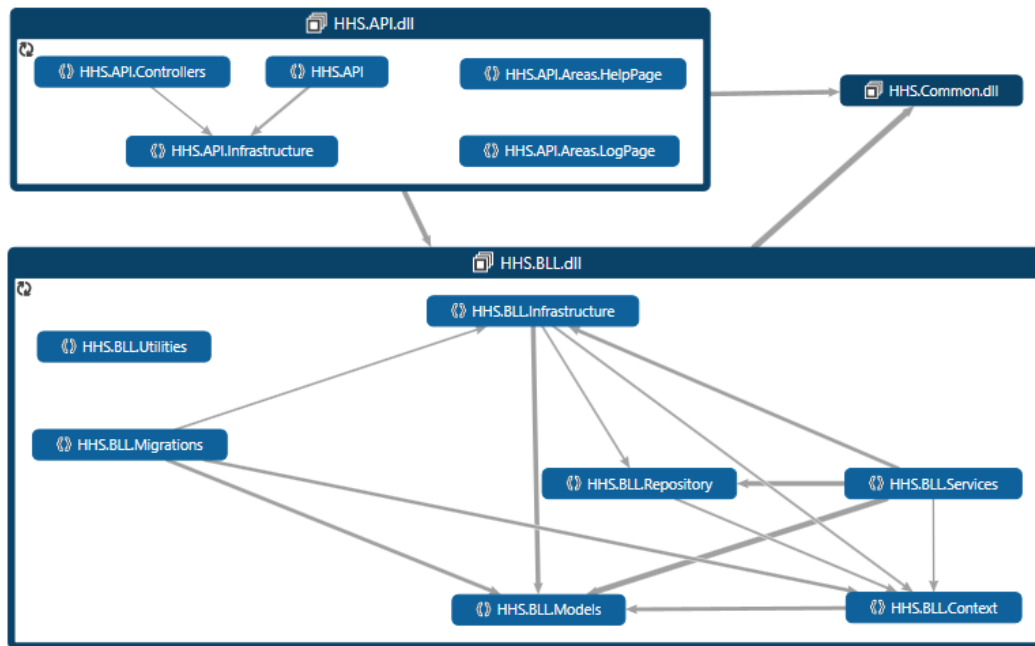


Figure 10: Server Dependency Diagram

**HHS.API:** At application startup, the `Global.asax` class is getting called which in turn initializes and configures the application. These configuration classes can be found in the `App_Start` folder.

**HHS.API.Infrastructure:** This is the core of the API. This folder contains all global `Http Filters` (see `Presentation Layer`). They process the exception handling, the input validation, the authentication, and enforce a secure `HTTPS` connection for each request.

**HHS.API.Controllers** The controllers contain all API routes like this one:

```

1 | [Route("api/user/{userId}/changeRole")]
2 | [HttpPost]
3 | public void ChangeRole(Guid userId, [FromBody] UserRole role)

```

The parameters can be defined as `URL segments` (e.g. `userId`), as part of the body (e.g. `role`), or as `query string parameters`. The controller actions directly call the

services (HHS.BLL.Services) and return the results. More detailed information can be found in the ASP.NET documentation <sup>12</sup> <sup>13</sup>.

**HHS.API.Areas** The Areas folder contains the help page and the log page. Both of these pages are only for developing purposes. In production, they can be deactivated. This can be done by commenting out the RouteConfig and both AreaRegistration classes or by simply setting the [NonAction]-Attribute.

**HHS.BLL.Migrations** The migration is the Entity Framework database versioning tool. All migrations are stored in this folder.

**HHS.BLL.Context** Entity Frameworks unit of work and repository is called Db-Context. Every database access is realized via the context.

**HHS.BLL.Models** The models are the object-oriented mapping of the database tables. On DB queries, such a models are returned.

**HHS.BLL.Repositories** The repository is a powerful class, that represents another abstraction layer on top of the Entity Framework context. It covers around 90% of all database access. For example insert, update, delete, and a very useful query function. Another benefit of the repository is the included logging capabilities.

**HHS.BLL.Services** All of the business logic is placed within the services.

**HHS.BLL.Infrastructure** This is generally the place of the DB model to DTO mapping configuration. It also contains some security utilities and the business logic layer initialization class.

## 5.5. HHS.Common Project

The HHS.Common project is used by both .NET applications. The server and the administration client.

**Responsibilities** The common project is a collection of code that is shared between solutions. It contains data transfer objects (DTOs), utilities, the logger and Retrofit.NET for communication.

---

<sup>12</sup><http://www.asp.net/web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api>

<sup>13</sup><http://www.asp.net/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2>



**NLog Logger** NLog is a powerful and configurable logger. As default we defined a file-logger, which is used by the admin client on production. For more flexibility, we created the opportunity, to define any other config file with different logging rules. Just call this on application start:

```
HHS.Common.Logging.NLog.NLogLogger.InitializeNLogLogger("myConfigPath");
```

Please consider that this takes around two seconds to execute, so the method should only be called once. This feature is used on the server to use different logging rules in development and production.

**Retrofit.NET** On the Android client, we use a nice web client called Retrofit. To provide developers a comfortable and common look and feel of web clients, we decided, to implement a similar one for .Net. Thankfully, Jordan Thoms already had the same intuition and implemented a Retrofit for .NET called simply Retrofit.NET<sup>14</sup>. As the backend web client he used RestSharp which is not under development anymore. According to several articles, RestSharp still has some bugs that can lead to unexpected behaviour. This is the reason, why we implemented a Retrofit.NET web client based on Microsoft's `HttpClient`. Additionally, we extended Jordan Thoms' solution with asynchronous messages.

The adoption of Retrofit.NET speaks for itself.

1. Create an API interface:

```
1 public interface IHSService
2 {
3     [Post("api/thread/{threadId}/post")]
4     void CreatePost([Path("threadId")] Guid threadId, [
5         Body] PostDTO post);
6
7     [Get("api/thread/{threadId}/post/{postId}")]
8     Task<PostDTO> GetPost([Path("threadId")] Guid threadId
9         ,
10        [Path("postId")] Guid postId);
11
12    [Post("api/thread/queue")]
13    Task<IEnumerable<ThreadDTO>> GetThreadQueue([Body]
14        Filter filter,
15        [Query("dynamicFilterValues")] string
16        dynamicFilterValues = "");
```

---

<sup>14</sup><https://github.com/jordan-thoms/Retrofit.Net>

```
14 | } // public interface IHSService
```

2. Call:

```
1 | IHSService hhsService = new
2 | RestAdapter("https://hhs.helvetas.com/").Create<
   | IHSService>();
```

The RestAdapter creates a proxy of the interface which handles the whole communication (see RestInterceptor). The proxy is generated with means of the library Castle.Core.

## 5.6. Administration Client

The Windows admin client is used by experts and admins. They proceed the tickets, create a Knowledge Base entry or send an alert to the users.

**The View-ViewModel Binding** While Model-View-ViewModel is an integral conceptual part of WPF, it is not in WinForms. Nevertheless, it is one of the most useful concepts for presentation layers. In HHS, the entire business logic is part of the server. HHS.Admin represents a thin client. The view models contains as much as possible of the presentation logic (this is not possible if a component does not offer a binding function).

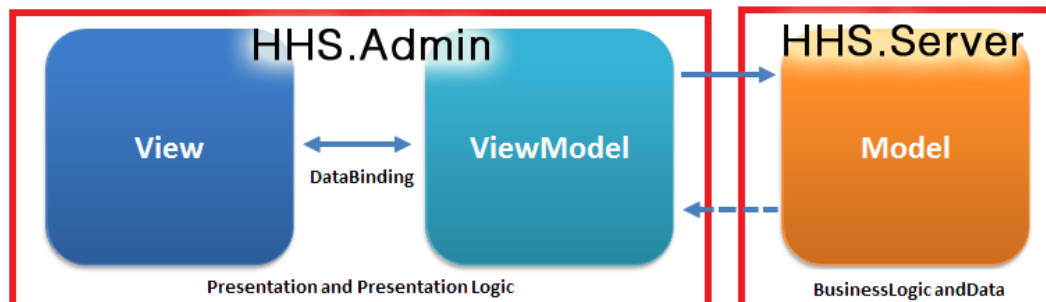


Figure 11: View-ViewModel-Model

15

<sup>15</sup>[https://de.wikipedia.org/wiki/Model\\_View\\_ViewModel](https://de.wikipedia.org/wiki/Model_View_ViewModel)

The following system sequence diagram (SSD) shows how we translated the MVVM pattern to WinForms:

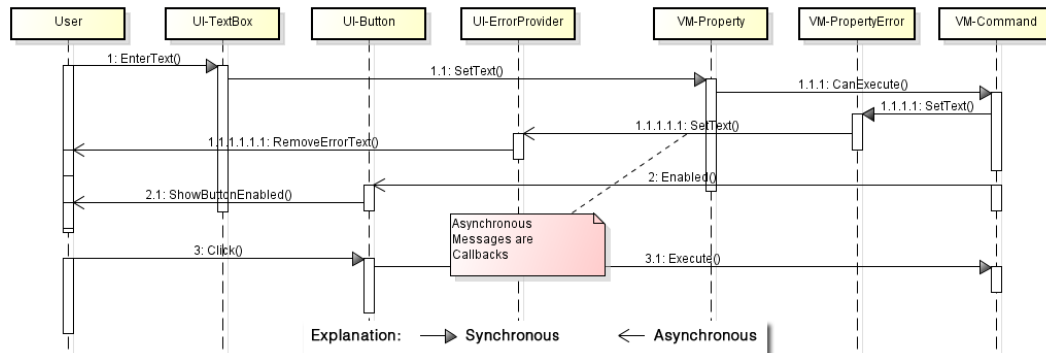


Figure 12: SSD UI-Command Binding

In the diagram, a case is shown when a user previously entered an invalid input, got an error message and sees a disabled button (initial state). Now, he types a valid input (1) which updates the view model property (1.1) and calls the `CanExecute()`-method (1.1.1). Because the input is valid, the error property in the ViewModel is set to an empty string (1.1.1.1). Over the property changed event of the property error, the error message is removed. Concurrently, the view model command raises a property changed event (2) which is caught by the view. Now, the button is enabled (2.1). Afterwards, the user can click the button (3) and execute the command (3.1).

Wherever possible, we have tried to make an automatic binding. On `DataGridViews` (tables), `TextBoxes` and `Labels`, we created a binding like this:

```

1 | this.locationTextBox.DataBindings.Add("Text", this.loginVm,
2 |   "Location", false, DataSourceUpdateMode.OnPropertyChanged);
  
```

This means that `this.locationTextBox.Text` is bound to `this.loginVm.Location`. The `OnPropertyChanged` update mode causes an update of the view if the location property is updated in the background and fires a property changed event.

A button should never be active if the action fails because of an invalid user input. That is why we used the error provider to show input errors and enabled an action button only if the `CanExecute()` method indicates success. Unfortunately, since the error provider and the `Enabled` property of a button are not bindable, we are forced to handle these steps manually in the view.

Going through the login form serves as a good example.

### 5.6.1. Dependencies

The dependency diagram is simplified for a better understanding and only shows the dependencies between the important components. References to utilities and helper classes have been removed.

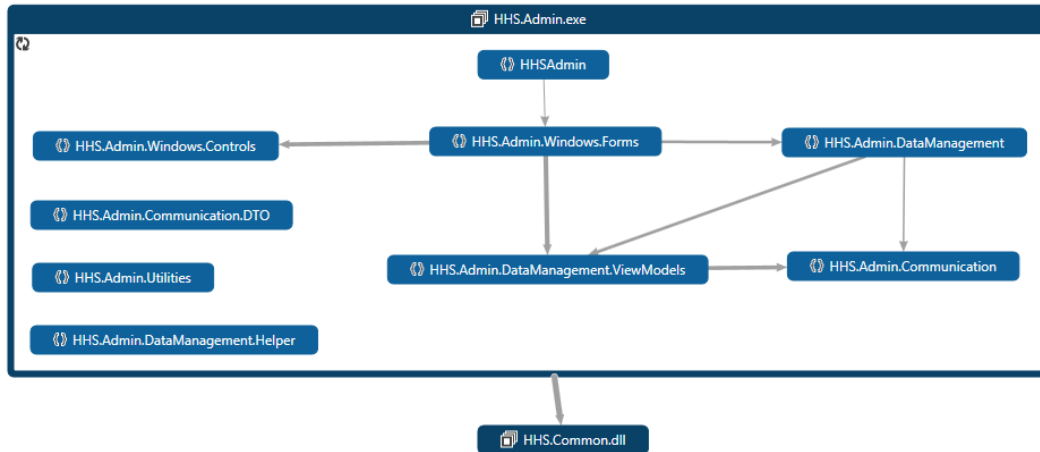


Figure 13: Admin Client Dependency Diagram

**HHS.Admin.Windows:** The Controls folder contains custom UI-components. In the Forms folder you can find all WinForms separated by navigation categories.

**HHS.Admin.DataManagement:** Mainly contains the view models.

**HHS.Admin.Communication:** Here are the web service interfaces defined. The admin client uses the HHS API and Google APIs.

## 5.7. Android Application

The Android app is a regular Android application using the default SDK as the main API. It has been built against API version 16 (minSdkVersion) using build tools version 23 (buildToolsVersion), as per requirement issued by HELVETAS. Providing compatibility to such an old Android version requires the use of *support libraries* that backport functionality.

### 5.7.1. Activity Lifecycle Management

The app's general workflow is determined by the rules of the Android Lifecycle Management, are depicted in the flowchart in Figure 14.

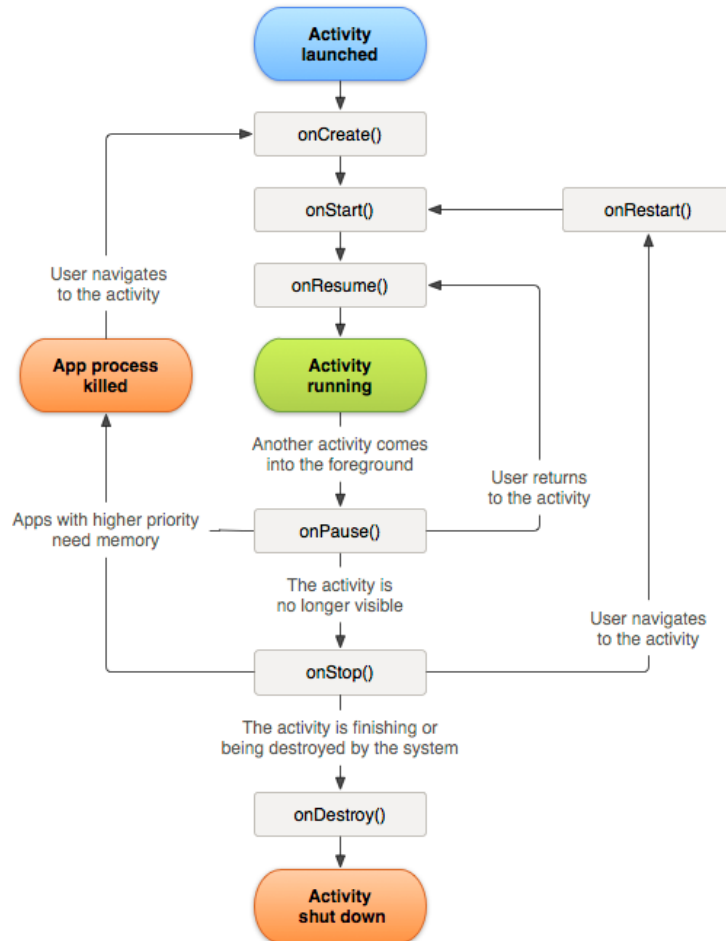


Figure 14: Android Lifecycle Flow Chart <sup>16</sup>

It is important to know that on Android, any app can be killed by the system at any point in time without previous notice. It is thereby suggested by Android design guidelines to save user edits whenever certain events occurs as context switches cannot be predicted.

The most important callback methods are `onCreate()`, `onStart()`, `onResume()`, `onPause()`, and `onStop()`, which is the place for services such as Google Play Services to be set up and destroyed, respectively.

Please refer to the official Android Documentation for further explanation<sup>17</sup>.

<sup>16</sup><http://developer.android.com/reference/android/app/Activity.html>

<sup>17</sup><http://developer.android.com/reference/android/app/Activity.html>

### 5.7.2. General App Architecture

The Android app follows the Android design guidelines and is based on Activities and Fragments. The Model View Controller (MVC) concepts are implemented. Models are represented by Active Records (as explained in Section 5.7.3) that are connected to views (i.e. layouts) via Activity or Fragment controllers. Unfortunately, Android does not thoroughly support data binding which makes data managements within controllers rather tedious.

### 5.7.3. Local Database / Data Model

The local database that is used in the Android app is shown in Figure 15. Database access will be explained afterwards in Section 5.7.5.

It is important to understand that data classes on the Android client, namely `UserData`, `ThreadData`, `PostData`, `FeedData`, and `ServiceData`, not only reflect the tables within the database but that they are also used directly as model classes at runtime using the *Active Record* pattern.

The tables are connected UUID primary keys that correspond to the UUIDs which are used on the server.

Android provides direct support for SQLite databases<sup>18</sup>. The OR mapper that is used in this project (see Section 5.7.5) will also use SQLite for storage. Unfortunately, SQLite provides little support<sup>19</sup> for advanced data types and can only store null values, integers, floating point numbers, strings and data blobs natively. For the purpose of serialization, converters had to be implemented that allow complex types to be serialized to types that can be managed by SQLite. These converters are defined in the package `com.helvetas.hhs.data.converters`.

---

<sup>18</sup><http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>

<sup>19</sup><https://www.sqlite.org/datatype3.html>

### 5.7.4. Database Diagram

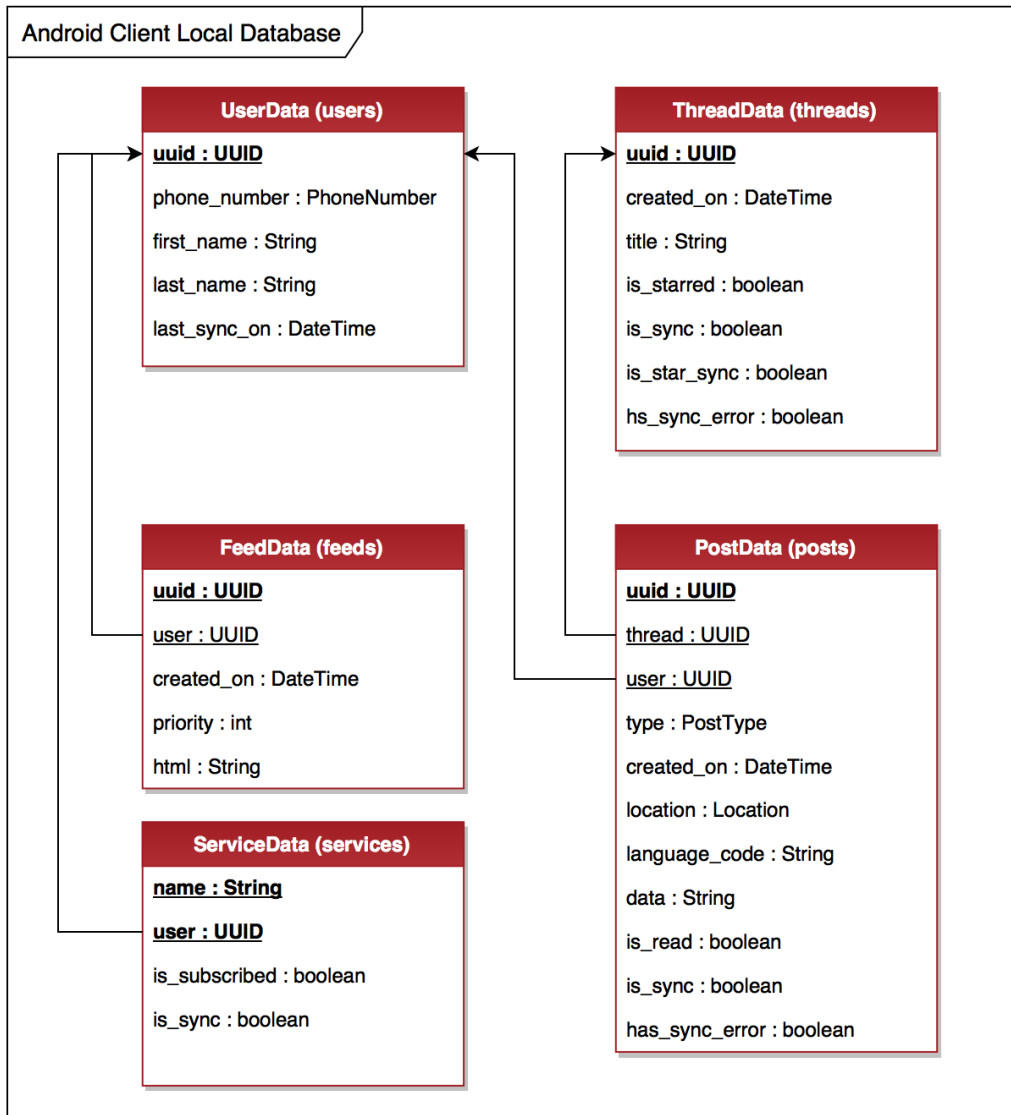


Figure 15: Android Client Local Database

### 5.7.5. ORM: Active Android

The Active Android library<sup>20</sup> is used for persistence. Active Android provides data access via the *Active Record* pattern. The implication of active records is that rows in a table can be treated as instances of a class, which greatly simplifies data management as these objects can and are used directly for the runtime data model.

<sup>20</sup><http://www.activeandroid.com/>

We have decided in favor of this approach because of the following reasons:

- Additional features can be implemented simply by creating a new model class that is able to hold the data which is needed at runtime.
- Object persistence (save) is fully automated and can be done on a fairly high level as SQL queries must only be written (using a builder) for querying the data, not for storing or updating records.
- The Android client, in strong contrast to the server, does not have to deal with massive amounts of data. In this situation, the additional object overhead can be dismissed in favor of ease-of-use.

### 5.7.6. Offline Mode & Synchronization

A specific requirement for the mobile app is the implementation of an offline mode with auto-synchronization. Synchronization processes will be triggered by either push notifications (Google Cloud Messaging<sup>21</sup>) or user events (Broadcast Receivers<sup>22</sup>).

As Figure 16 shows, the synchronization process is divided into three specific parts that react to different incoming events.

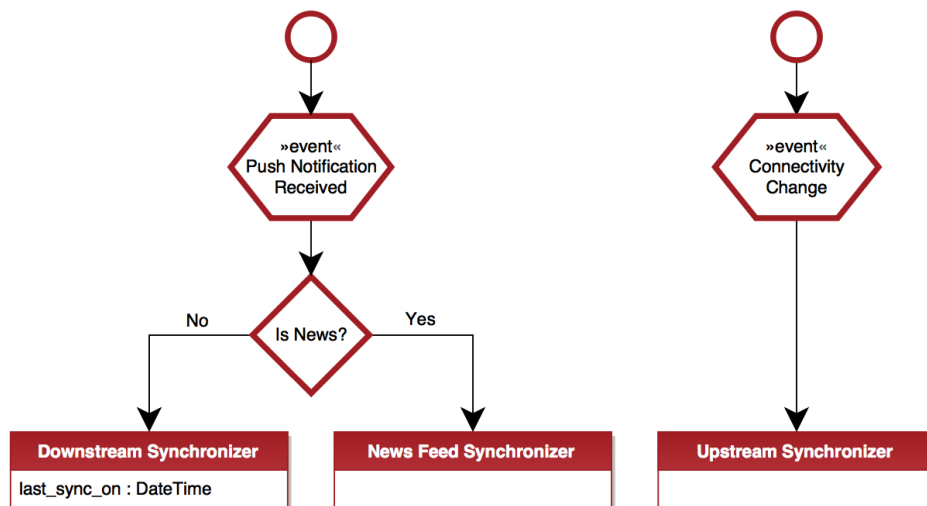


Figure 16: Android Client Local Database

<sup>21</sup><https://developers.google.com/cloud-messaging/>

<sup>22</sup><http://developer.android.com/reference/android/content/BroadcastReceiver.html>



**Downstream Synchronizer** is polling new threads and posts from the server when invoked, based on the date and time of last synchronization. Data is never polled twice in order to avoid unnecessary overhead.

**Upstream Synchronizer** sends all local changes to the server that have not yet been synchronized. The Upstream Synchronizer does not differentiate between threads and posts that are created while in Online Mode or Offline Mode. All local changes go through the Upstream Synchronizer.

**News Feed Synchronizer** is receiving and processing news feed entries that arrived via push notifications. Unlike threads and posts, news feed entries are sent directly and will not be polled.

### **Asynchronous Execution**

This division allows the logical separation of the synchronization processes into smaller parts that can be run individually. As mentioned before, the synchronizers will be executed by certain events that happen asynchronously to the program flow. In fact, events will trigger even if the app is not currently in use, thus it is ensured that the user will always receive latest updates as soon as possible.

### **App Message Dispatching**

Once a synchronizer completed the synchronization process, a broadcast message will be sent to internal receivers and user notifications will be displayed on the phone. In case the app is currently running, it will update its view models so that the user interface shows the new data (i.e. new messages and news).

## **5.8. SMS Gateway**

In order to satisfy the requirements involving sending SMS messages (i.e. sending alerts and activation messages), an SMS gateway must be configured. As HELVETAS was unable to provide us with such a service, we had to implement a custom “poor man’s” solution for development and testing.

The workaround consists of two parts:

**PHP Script** running on a remote server that accepts a phone number and a message via POST. Upon GET requests, the script returns the next SMS from its queue and deletes it after delivery.

**Android SMS Sender** that retrieves the stored message from the PHP script and sends it as an actual SMS to the recipient number via the phone’s SIM card.

The complete source code of the workaround SMS Gateway can be found in the appendices of this thesis (smsg). Do note that for further testing and especially for production, an actual SMS gateway has to be set up locally in Nepal.

### 5.9. Database

Usually, the domain model (Section 3.1) can be converted into the database model. Right after the Database Model, a short description of each table will follow. The tables correspond to the models in the namespace HHS.BLL.Models. Detailed information for each column/property can be found in the code documentation in the mentioned namespace.

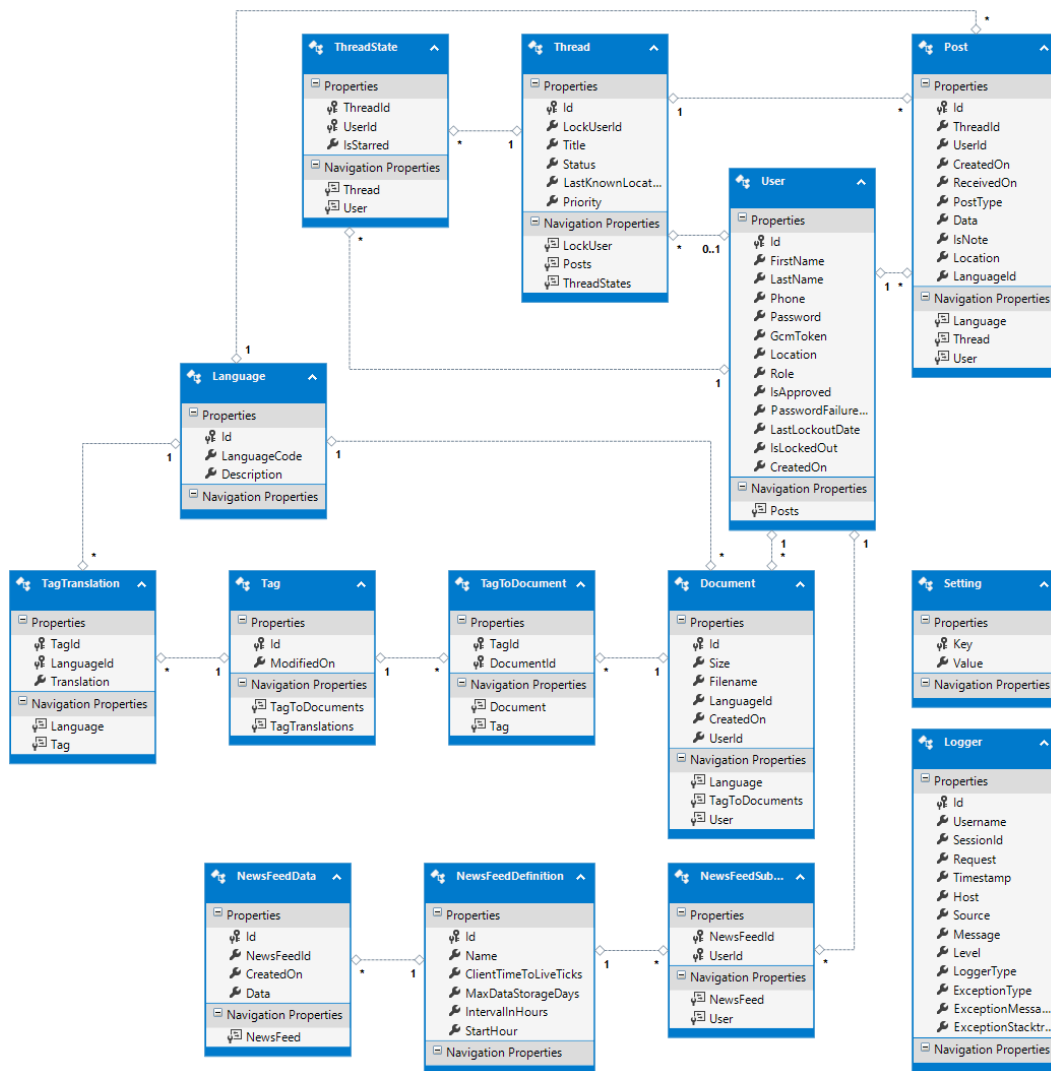


Figure 17: Database Model

**User:** Represents a user, an expert or an admin. An actual location for distance calculations and an active Google Cloud Messaging token for notifying updates are essential. The phone number must be unique because it is used for authentication.

**Thread:** Represents a set of multiple posts with the same subject

**Post:** Represents a single unit of a thread. A post could be a question or an answer. The post type indicates, whether the data field contains text, voice or an image. The received date is set by the server and is decisive for the post ordering.

**ThreadState:** Indicates whether a user has starred a certain thread. Further, this table might also be used to indicate whether a user has archived a thread.

**Language:** This table specifies which languages are supported by the HHS. Dynamic translations (e.g. TagTranslation) have to be made for each language.

**Tag:** A tag is an attribute of a document. Tags are used for searching documents.

**TagTranslation:** Represents a description of a tag in a certain language. A translation is required for each tag and language.

**Document:** A document may be a text document or a multimedia file. Documents are defined by tags.

**TagToDocument:** Represents the tag to document associations.

**NewsFeedDefinition:** All available news feeds/services are stored in this table.

**NewsFeedSubscription:** When a user subscribes to a news feed, it is stored in this table.

**NewsFeedData:** Represents a news feed data entry. With the help of this data, after a restart of the server can determine, if a news feed has already been executed or if there is still an open job. The stored data could also be used for analysis purposes. The idea is, that the data is deleted after a certain period.

**Setting:** All server settings are stored in the database.

**Logger:** This table is used by NLog.

### 5.9.1. Use of GUIDs

We have decided to use GUIDs (128 bit integers) for primary keys. This is to guarantee uniqueness across different tables and systems. This decision has been made in order to support distributed systems in the future. It also allows for the generation of IDs on different systems other than the target platforms where the data is to be stored.

*Do note that GUID (.NET) and UUID (Java) are equivalent terms.*

### 5.9.2. Table Names

Tables are named straight after their purpose, e.g. table *user* stores user information.

### 5.9.3. Field Names

To ensure a certain level of consistency within the database, we have decided to respect the following list of naming conventions while still allowing a certain level of freedom if it serves general readability / maintainability purposes.

Type	Suffix	Decision
ID / PK	Id	Table specific GUIDs that represent primary keys are called Id. The naming is clear and obvious enough.
ID / FK	Id	Foreign keys (also GUIDs) have a suffix “Id” that follows the name of the entity they are referencing, e.g. “UserId”.
DATETIMEOFFSET	On	Fields storing date and time information are always suffixed with “On” to indicate their purpose.
BIT	Is	Boolean fields that store a value of other <code>true</code> or <code>false</code> are prefixed with “Is”.

### 5.9.4. Field Types

This section describes the field types that are used in the database. We have decided that we will use basic types only and not attempt to create our own custom types. This decision has been made due to compatibility and scalability reasons as custom types might not always be supported and are hard to modify once they are used.

<b>Type</b>	<b>Description</b>
GUID	Globally Unique Identifier, 128 bit. This is the main type for primary keys and has been chosen with distributed databases in mind.
NVARCHAR	These types are used to store strings. Strings are always encoded with UTF-16 to maximize compatibility, especially with languages with special characters.
BIT	A field of type BIT stores either <code>true</code> or <code>false</code> . In MSSQL, boolean values are stored as a single bit (that will be expanded to one byte depending on other data within the table).
INT	Regular type to store integer numbers. This type is used to either store general numbers or enumeration values.
DATETIMEOFFSET	Stores information about a particular point in time. The time zone is stored along with this type to prevent conversion issues.
DBGEOGRAPHY	These type is used to save coordinates (latitude, longitude) as well as the altitude. The use of this datatype enables database supported location calculations.

### 5.9.5. Language Identifiers

Language codes are based on the ISO 639-1 and ISO 3166-1 standards. The value is equivalent to a five character string in the form “ll-cc”, where l indicates the language code and cc the country, e.g. “en-us” for American English.

## 6. Concepts

This section shows the principles and concepts of the business domain that were applied to the HELVETAS Horticulture System.

### 6.1. Ticket System

To optimize the communication between users and experts and to decrease the response time, the ticket system was developed. In contrast to traditional ticket systems that works with e-mails, the HELVETAS system should be organized as a chat app on an Android client. Nevertheless, the HHS ticket system is influenced by many concepts of traditional ticket systems.

Meeting all requirements was a big challenge. We had to pay attention on low data rates and the offline use of the application on Android phones. The process of the ticket system is explained by means of communication diagrams. Both diagrams show the technical implementation of the use cases *Help Query* and *Process Ticket*:

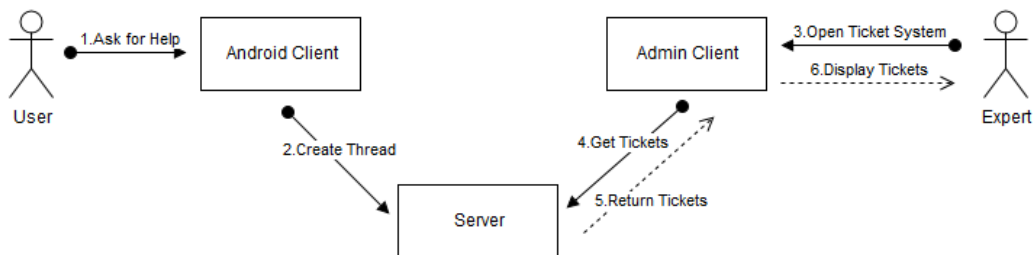


Figure 18: UC Help Query - Communication Diagram

As reminder: The terms Thread and Ticket are used interchangeably. The diagram shows the communication between the android client, the server and the admin client. As we see, the admin client polls the tickets.

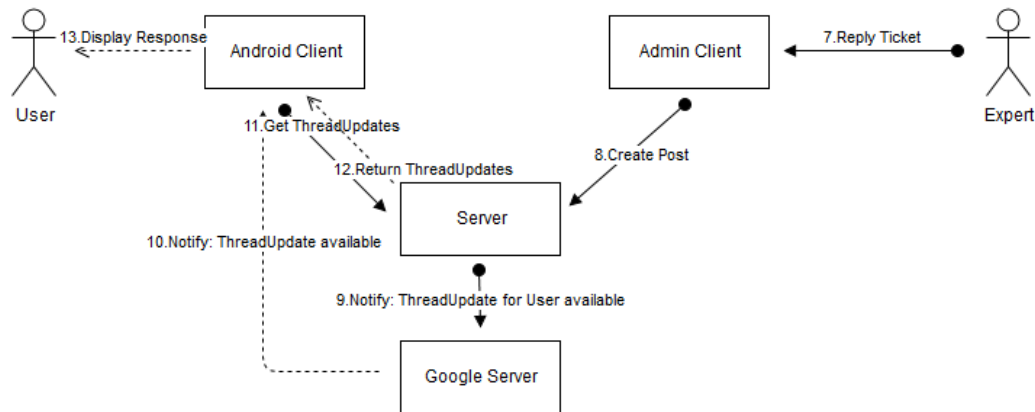


Figure 19: UC Process Ticket - Communication Diagram

At first sight, the procedure might look confusing. But the extra step over the Google Cloud Messaging server is required to enable push notifications. This approach saves a lot of traffic and battery that would otherwise be needed for polling. As soon as the Android client receives the push notification from Google, it requests the new posts from the server.

### 6.1.1. Thread Queues & Thread Filters

The initial idea behind the thread queues was the ability to filter threads by different criteria on the admin client. The proximity queue should show the queries of the nearest farmers; the locked queue shows all threads that are marked for a particular expert; the starred queue shows the favorite threads. To provide the opportunity of dynamically building such queues at runtime (i.e. without recompiling the source code), we introduced the threads filters (see use case Create Ticket Filter [item 4.1.3])

To generate dynamic thread filters, we needed the opportunity to build database queries at runtime. According to a post on Stack Overflow<sup>23</sup>, there mainly are three opportunities:

- Using Linq's composable behaviour is too inflexible for our requirements because with this solution, every property that could be in the where clause, would have to be hardcoded.
- Dynamic Linq provides a Linq extension that allows passing a string instead of an Function-Expression. This solution proved to be extremely flexible and seemed to work, but unfortunately geographic functions like distance were not

<sup>23</sup><http://stackoverflow.com/questions/5139467/how-to-create-linq-query-from-string>

supported. The exception “Methods on type DbGeography are not accessible” was raised.

- That lead us to the third solution: Building plain old SQL statements. Additionally, it should be said that SQL injection is no problem in this solution as filters are only accessible to experts anyway. Since all experts have rights to see the entire data, there is no remaining abuse case.

To define these dynamic SQL queries, we introduced a “Expression-Term-Factor” like grammatic. For more complex constructs, a filter rule can contain a list of filter rules. While parsing, this list is put into parentheses.

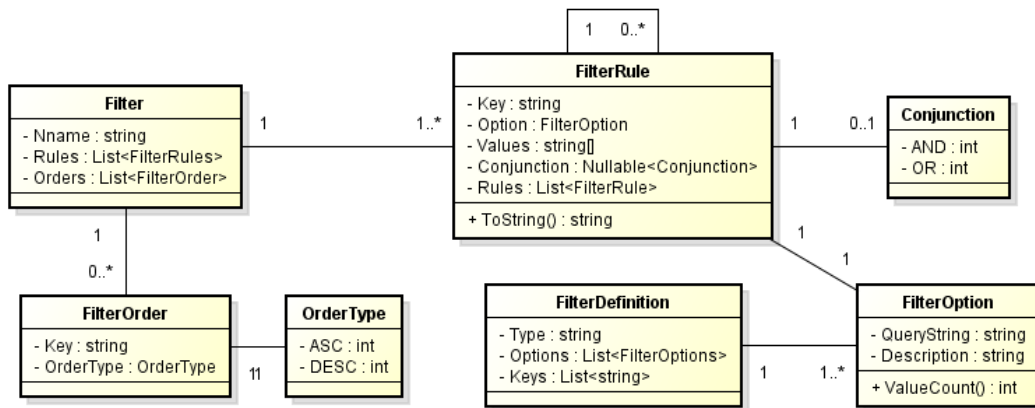


Figure 20: ThreadFilter Class Diagram

The filter system is not completed yet. To demonstrate the current state and the final meaning we created an SSD:

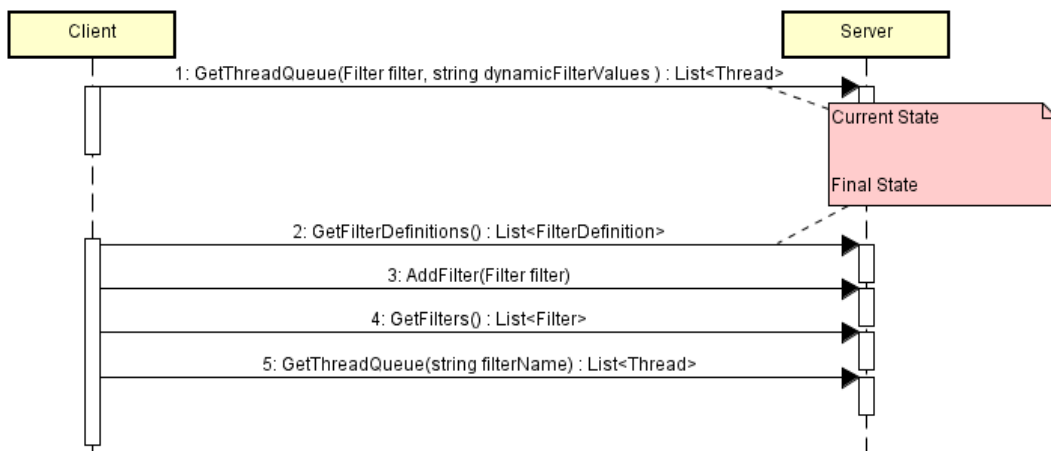


Figure 21: SSD ThreadFilter



Currently, the filters are held on the clients and are sent to the server on each queue request (1). The sending of the filter in the body is only a temporary solution (also, it is not REST-conform). In the future, it will have to be more modified and extended. The client asks the server for the filter definitions, which contains all supported keys and their options (2). Due to this information, the client is able to create a filter. Filters can be added (3), updated and deleted, and will be managed on the server. The client asks the server for all filters (4). It then requests each queue with a single request (5). The request will only contains a filter name.

The final state should also contain a user interface on the admin client, which allows the creation of these filters without much technical know-how (see the image in Use Case *Create Ticket Filter*).

## 6.2. Location Determination

The connection of widely spread-out people is a central point in the vision of the HELVETAS Horticulture System. The idea is that a farmer can query the nearest expert. To provide information of proximity, we had to work with the users' locations. On each application start, the client tells the server the current user location.

On Android, the location can be requested using Google Location Services. The provided location is the last known location of the device, so GPS must not explicitly be queried as coarse locations are sufficient.

Windows systems generally do not come with a GPS receiver installed. Even though it is sometimes possible to query the location of a laptop user, this approach is too unreliable and almost certainly does not work for desktop computers. As we cannot expect location information to be available, we had to come up with a different solution.

When an expert logs in, he has to tell the system his current location. Over the Google Geocode and Google Elevation APIs, the coordinates and altitude can be requested. If a valid location is found, the user position is sent to the server. The proximity calculation depends on this position.

### 6.3. Knowledge Base

At the beginning of the project, one of the most obscure requirement was the Knowledge Base. HELVETAS did not have a concrete idea how the KB should look like nor how it should be structured. It was thus also part of this bachelor thesis to come up with a flexible and generic structure that can be customized to different domains.

#### 6.3.1. KB Structure

We suggested to differentiate between System Base (i.e. Ticket System) and the Knowledge Base. The Knowledge Base should contain only documents with a high informative value.

System Base	Knowledge Base
Includes <ul style="list-style-type: none"> <li>• User Management</li> <li>• Threads &amp; Posts</li> <li>• Alerts &amp; Services</li> </ul>	Includes <ul style="list-style-type: none"> <li>• Case Studies</li> <li>• Best Practices</li> <li>• How-To Guides &amp; FAQ</li> </ul>
Structure <ul style="list-style-type: none"> <li>• Semi-Structured (by user, thread, thread title)</li> <li>• Searchable</li> </ul>	Structure <ul style="list-style-type: none"> <li>• Tags</li> <li>• Language</li> <li>• Searchable</li> </ul>
Visibility / Permission <ul style="list-style-type: none"> <li>• Expert: Read/Write all</li> <li>• User: Read/Write own Threads</li> </ul>	Visibility / Permission <ul style="list-style-type: none"> <li>• Expert: Read/Write all</li> <li>• User: Read all</li> </ul>

#### 6.3.2. Tags

Initially, there was the idea that the documents could be structured the same way as a gardening calendar. The result would be a two-dimensional matrix with the axis *Crops* and *Season*. This structure is overly fixed and thus many documents would not fit in. For example the user guide of the HHS could not be placed in such a structure. If we consider the generic system approach, we see, for example, that specifying *Season* would be useless in health care. These thoughts lead us to

the dynamic structure with tags and language. The language can be regarded as a required tag. With the tag structure, we come from a 2-dimensional to an n-dimensional approach, because a document can have as many tags as is appropriate.

### 6.3.3. Search Mechanism

If a user wants to find a document, he specifies the search tags. He enters a tag name (1); the system then gives tag propositions in a combo box. If the user selects an item of the combo box, the tag appears in the tag list (2). If the user presses the search button, a request with the apple, tree and fungi tags is sent to the server. As result, the client receives a list of documents with following information: Size, filename, language, tags of the document, and accuracy (4.). The timestamp and the user that created the document is only sent to experts.

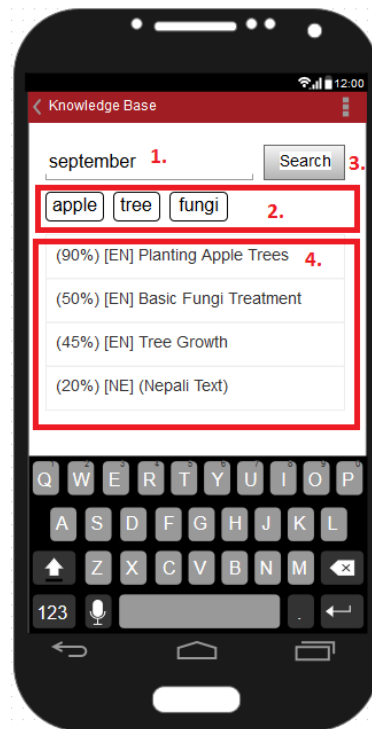


Figure 22: Knowledge Base: Android Search

### 6.3.4. Accuracy

We realized that good search results are the A and O of a file store. As the tags provide us a fine-granular structure of the documents, we are able to calculate the accuracy. Our accuracy calculation works with the following principles:

- A document with a high document tag match count is more accurate than a document with a low document tag match count.
- A document with many specified tags is less accurate than a document with just a few tags.

To understand the calculation, we have to understand following terms:

- TotalDocTags is the tag count of a document, e.g. apple, tree, planting, spring.
- TotalSearchTags is the tag count of the search, e.g. apple, tree, fungi.
- Matches is the accordance between TotalDocTags and TotalSearchTags. In our example the match is two.
- DocRatio is the ratio between total doc tags and matches. In our example:  $1.0/TotalDocTags \times Matches = 1.0/4 \times 2 = 0.5$
- SearchRatio is the ratio between total search tags and matches. In our example:  $1.0/TotalSearchTags \times Matches = 1.0/3 \times 2 = 0.67$

The challenge was to find a good function to combine the DocRatio and the SearchRatio. First, we made tests with the exponential function:

- $Accuracy = DocRatio^{SearchRatio}$ . The problem with this approach is that a search query with more search tags is preferred to a query with less search tags and equal matches.
- $Accuracy = SearchRatio^{DocRatio}$ . Here, we have the contrary problem. A document with more tags is favored to a document with less tags and equal matches.
- $Accuracy = (DocRatio + SearchRatio)/2$ . This is the current solution. Unfortunately, it has a less beautiful exponential curve than the other calculations. Nevertheless, the logic is correct.

### 6.3.5. Localization

The Knowledge Base concept is also characterized by the multi-language requirements. No matter whether a user searches in English or Nepali, he will get the same search results. The reason is, that each tag is translated. There are not two tags with the meaning of apple (one in English, one in Nepali). There is only one tag and

documents are assigned to this tag definition. The question may arise: why does the system show English documents if the user searches in Nepali? The consideration is, that it is better to show any results instead of no results. Also, it is better to read a more accurate guide in English than a less accurate guide in Nepali. If the user is not familiar with the other language, he could probably query an expert for translation. Otherwise, if there are too many results shown, the user is supposed to make a more specific search.

### **6.3.6. Concurrency**

A further challenge related to distributed systems is the concurrency aspect. How can the system guarantee that changes of a user are not overridden? One solution might have been document versioning. While this would have been a viable solution, it would have required too much time while bringing too little benefit. Our final solution is simpler, yet serves its purpose rather elegantly. There is no update functionality available. That implies that every upload results in a new document. After that, the user can delete the old version of the document. In this way, we could bypass the problem.

### **6.3.7. Upload Issues with Large Files**

Having established the Knowledge Base feature, we discovered a problem with uploading large files. After investigating, we could localize the problem: the default IIS maximum request length of 4MB has been exceeded<sup>24</sup>. This point is important to know for future developers. With the `httpRuntime / maxRequestLength` and the `requestLimits / maxAllowedContentLength` attributes in `Web.config`, the maximum request length can be controlled. Because HELVETAS did not provide us with concrete size limitations, we have set the default limit to 1GB. This allows larger files such as videos to be uploaded. In case the requirements change, `Web.config` will have to be adjusted.

## **6.4. News Feed - Services**

The definitions of News Feed and Services are described in Section 3.2 (Domain Terminology). As is apparent in the requirements and the use cases, user shall be able to subscribe and unsubscribe to and from several services. They should periodically receive updates for their news feeds. In the scope of this bachelor thesis, we have implemented two sample services: Weather Forecast and Market Prices.

---

<sup>24</sup><http://stackoverflow.com/questions/3853767/maximum-request-length-exceeded>

### 6.4.1. Weather Forecast

The weather forecast service was a really hard nut to crack. HELVETAS suggested us the OpenWeatherMap API (see <http://openweathermap.org/price>). As we worked with the free version, we have come across limitations: at most 60 calls per minute; at most 50'000 calls per day. Around 10'000 users are estimated for the HHS. If all users were subscribed to the weather forecast service, the weather update job would take about two and a half hours – every day. Because this behaviour is unacceptable, we had to find a solution.

### HAC: Hierarchical Agglomerative Clustering

We considered it would be useless to perform two separate API calls, if me and my neighbour were subscribed to the service. In this case, we could combine the requests to one. This thought was suggesting to find a representative subset of all subscribed users. Then, we would only have to process a forecast request for each member of the subset. As we were familiar with the principles of clustering (studied in the information system module), we found the HAC algorithm during research.

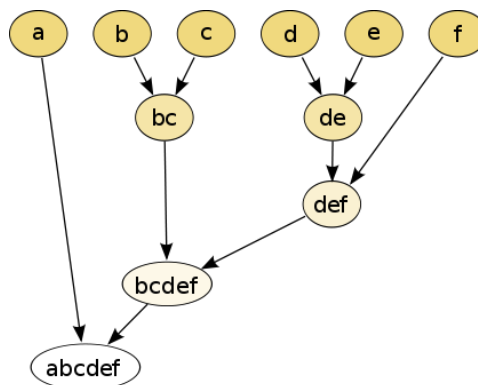


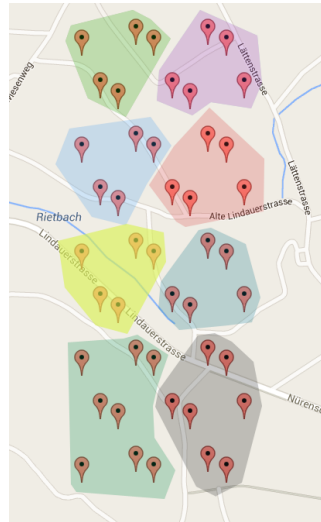
Figure 23: HAC diagram <sup>25</sup>

This graphic shows that on each iteration the two nearest points are grouped.

<sup>25</sup>[https://en.wikipedia.org/wiki/Hierarchical\\_clustering](https://en.wikipedia.org/wiki/Hierarchical_clustering)

We defined a configurable variable `maxWeatherApiCalls`. If the subscribed user count is less than this variable, the weather job makes one API call for each user. Otherwise, the HAC-algorithm groups the subscribed users to clusters and makes one API call for each cluster.

We have tested the correctness of the algorithm. The visual test result can be seen in following image:



*Figure 24: HAC algorithm test*

## Performance Problems

As a template we have used a badly documented HAC algorithm from semanticquery.com [5] which was written for text document clustering. Because our initial working algorithm was extremely slow, we had to tune it up. Unfortunately, the readability suffered by the optimization. This is the reason, why you can find two versions of the algorithm in the code: The slow one, that is easier to understand, and the fast one, with less readability.

Starting with 1'000 locations, the calculation of 900 clusters initially took 628 seconds. After using an integrated coordinate-distance-calculation, we sped it up to 220 seconds. This lead us to the assumption, that the most expensive part of the algorithm is the distance calculation. In the final version, we stored the distance calculations and recalculated only the distance to the newest cluster (this solution is also implemented in the sample from semanticquery.com). The new calculation of 900 clusters only took 8 seconds. The calculation of 100 clusters took 23 seconds.

Currently, we make a bottom up (agglomerative) calculation. The iteration count

could be increased in the example with 500 clusters, if we would transform the algorithm to a top down approach (divisive). For large numbers of users, the algorithm requires a lot of memory. In our tests, we managed to reach a maximum of 13'000 location before we started receiving `OutOfMemoryException` exceptions.

### **Distribution**

For distribution, we could benefit from the previously used Google Cloud Messaging. GCM provides a topic messaging that could be used. Unfortunately, the weather forecast service will be individual for every user and thus cannot be aggregated in topics (channels). GCM does provide multicast capabilities, however, which could be used for distributing messages to all users within a the same cluster.

#### **6.4.2. Market Price**

For the market price service, there was no API available. This fact means that we had to improvise and come up with a workaround for receiving the required data.

- Until this point, we could handle every request as `application/json` calls. The weather forecast API, Google Cloud Messaging, Google Geocode, Google Elevation, the SMS server API and, of course, the HHS API, are all supporting JSON. For the market price, however, we had to use `application/x-www-form-urlencoded` calls on a regular website. We were obligated to extend the Retrofit.NET with this media type.
- The second, rather tedious point is that “kalimatimarket<sup>26</sup>” does not provide an API at all. We were forced to parse malformed HTML that we had to pre-process first before we could access relevant data.

Once the information has been extracted, it could be sent to subscribed users over GCM multicast.

#### **6.4.3. Quartz.NET – Job Scheduler**

After a comparison of several job schedulers, we decided for Quartz.NET, a proven framework <sup>27</sup>. We decided to implement one background task per service. This approach gives the opportunity to set an individual frequency for each service.

During our tests, we recognized, that the IIS webserver periodically recycles its pro-

---

<sup>26</sup><http://kalimatimarket.com.np/daily-price-infomation>

<sup>27</sup><http://www.hanselman.com/blog/HowToRunBackgroundTasksInASPNET.aspx>



cesses (responsible for the recycling interval is the *Regular Time Interval (minutes)* property in the advanced settings of the Application Pool). To avoid skipping over a news feed job while recycling the processes, we implemented a security routine. On each server startup, this method checks whether a certain job should have been run. If the news feed job has not been started in time, it will be started immediately.

#### **6.4.4. Extensibility**

We designed the news feed feature with the the generic idea in mind. Following configurations can be set for a new service:

- `ClientTimeToLive`: Indicates the timespan of which a news message should be delivered. If the client does not receive the message within the timespan, the message is deleted. This option is very useful since news messages lose their relevancy after a certain time.
- `MaxDataStorageDays`: Indicates the number of days the data is stored in the database. After this periode, the data will be deleted. This option helps controlling increasing data volumes.
- `IntervallInHours`: Indicates the interval in which the news feed gets refreshed.
- `StartHour`: Indicates the time of day when the job is executed the first time.

## 7. Results

The results of this bachelor thesis are described in this section, along with further information on project status, known issues, and future development.

### 7.1. Achievements

During the course of this project, the HELVETAS Horticulture System has been designed from ground up according to the initial input of HELVETAS Nepal and our own ideas and concepts.

We have thoroughly analyzed the current situation in Nepal and then initiated the design process of a ticket system prototype that will be used to streamline communication between HELVETAS experts and local farmers.

The ticket system consists of three distributed applications that have been created within the last fourteen weeks, namely the Back End Server, the Windows Administration Client and the mobile Android application. These three applications are all part of a large and complex system that has been elaborated in this bachelor thesis.

All three applications are multilingual and currently offer translations for English and Nepali.

#### 7.1.1. Achievement: Server

The server is fully functional and implements most of the requirements that have been specified at the beginning of this project. It provides the necessary means for communication between instances of the Administration Client and a large number of mobile Android clients.

The server implements much more than the basic functionality that is required for a ticketing system. It not only correctly manages users, threads and posts, but it is also able to collect data from third party sources such as weather forecasts and market prices that will be forwarded via a subscription model to target users. It runs complex spatial calculations based on user locations that allow distance-aware queries to be executed.

With the integrated Knowledge Base, the server is able to store an arbitrary amount of documents that can be accessed via the clients and thus provides farmers with vital information.

Using the HHS RESTful API, clients are able to easily and effectively communicate with the server over a secured connection.

### **7.1.2. Achievement: Administration Client**

The Administration Client offers basic functionality for experts to process tickets and engage with their users. As a thin client, it primarily serves as the server's front end that allows admins and experts to make changes to the data records.

It provides the necessary functionality to filter threads according to certain criteria, such as proximity, and to send messages back and forth between experts and farmers. The admin client lets experts search the Knowledge Base and upload new files to it, thus adding value.

The program can be distributed easily with a self-extracting archive that allow it to be installed on any computer that is meeting its requirements.

### **7.1.3. Achievement: Android Application**

With the Android app that will be distributed to local farmers in Nepal, they will be given the opportunity to instantly ask an expert for advice via an easy-to-use app. In an optimal scenario, the farmer will receive competent help within a couple of minutes.

Due to the fact that Internet connectivity is not always available, sophisticated synchronization capabilities have been implemented that allow the farmers to continue using the app even when they are currently offline. The app allows farmers to create new threads, attach text and photo messages to existing conversations, and subscribe to different services. The client features a news feed that displays important information such as weather forecasts, market prices, and alerts.

In order to save bandwidth (and thus high costs), the application only communicates with the server when it absolutely needs to. New posts or news feed entries are signaled with push messages originating from the server and displayed to the user immediately.

## 7.2. Open Tasks

The following list is a selection of open tasks that could not be addressed during this project. These task shall serve as a starting point for future development.

### 7.2.1. Server

- Implement an actual SMS gateway.
- Implement correct pagination for large result sets.
- Extend filtering capabilities.
- Either extend or remove thread priority specification.
- Implement localization for news feed messages.
- Extend Admin Client with CRUD capabilities where needed, e.g. implement deletion of tags.
- Find better third party solutions for market price service.

### 7.2.2. Administration Client

- Improve error handling and error feed back.
- Display licenses in about dialog.
- Implement server search to deal with large result sets.
- Threads can currently not be assigned between experts.
- Implement preview functionality for photo messages.
- Implement additional CRUD functionality where required.

### 7.2.3. Android Client

- Improve error handling and error feed back.
- Delete news feed entries if too old.
- Implement Knowledge Base search.
- Improve multi-user management or revert back to single user.

#### **7.2.4. Use Cases**

- Find Specific Ticket / Find an old Thread.
- Process Ticket Extension 3b, 5a (Manually send message to several farmers for more information).
- Create Ticket Filter.
- Alerts.
- Analysis.

### **7.3. From Development to Production**

In the following list are a number of issues specified that need to be addressed before going into production. Please also consider the guides that have been attached to this project which can be found in separate files in the appendices.

#### **7.3.1. Server**

- Install valid SSL certificate.
- Check all settings in database (hhsdb.Setting, hhsdb.Language).
- Setup actual SMS gateway. The current workaround is not adequate enough to be used for production.
- Access market prices from a source that is providing an API.
- Consider purchasing a license for Open Weather Map to increase data allowances.

#### **7.3.2. Admin Client**

- Sign the application with a certificate (HHS.Admin Project, Signing).
- Remove deactivation of SSL certificate verification (HHS.Admin.Program.cs).
- Check client settings (Settings.settings).

### **7.3.3. Android Client**

- Once a valid SSL certificate has been installed on the server, disable the development SSL hack.
- Generate a certificate for signing apps in release mode.
- Once finished, distribute the app via Google Play.

## **7.4. Lessons Learned**

- Project Management: even small thing may take a large amount of time if anything goes wrong. This fact should be reflected in the planning to an even larger extent.
- For Android, try to stick with core Android libraries instead of relying on too much third party code.
- Try to incorporate better error handling from the start.

## 8. Project Management

This sections gives insight into the project management of this bachelor thesis.

### 8.1. Time Management

Feature - Work	Week	Target Time	Actual Time
Analysis, Domain Model, Writing the project request Setup Project, Getting familiar with the environment and frameworks.	1-2	104h	119h
Setup entire Infrastructure for Android Client, Windows Client and Server	3-4	104h	542h
Ticket System: Concept, distribution, implementation backend & GUIs, tests, documentation	5-6	104h	-
Knowledge Base: Concept and structure, search algorithm, implementation, tests, documentation	7-8	14h	37h
Services: Implement weather forecast and market price	9	52h	28h
Alerts	10	52h	-
Analysis and buffer time for miscalculations	11-12	104h	-
Documentation: Usability tests, scalability tests, write user and admin guides, technical report	13-14	104h	76h
	<b>Total</b>	<b>728h</b>	<b>802h</b>

In the table above, we see the differences between the initial time planning as target time and the real time needed as actual time. Most of the features were well planned. The only point, where our prediction failed, was the setup of the infrastructure. During the project, we experienced first hand how much time the initial phase of a project can require. Please note, that it was not possible to split up the time for the ticket system and the infrastructure, because they merge into one another. Firstly, we underestimated the full amount of work, that is required for the whole infrastructure of three different applications. On the other hand, we underrated the implementation time for several functions.

Knowledge Base and Services took much less time than estimated. This is an indicator that shows how much evolved the prototype is – especially the server part. The implementation of these new features could be executed very efficiently and quickly.

The 12 ECTS demands a total of 360 hours per student to work on the project.

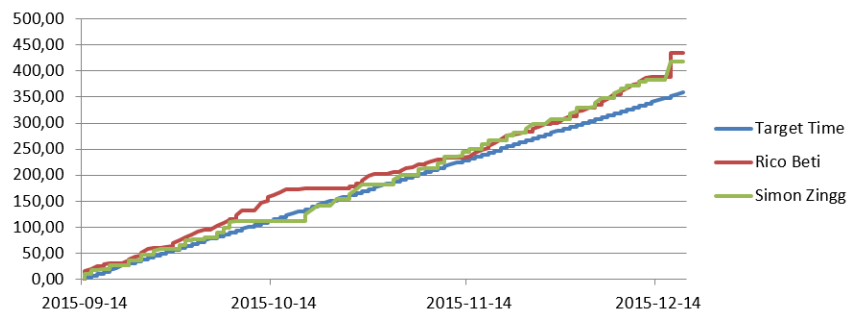


Figure 25: Time per Student

As the diagram above shows, we had a good time scheduling. From the beginning of the project, we worked the demanded time for the project. Toward the end, we worked more than 4 hours every day. The result is a plus of 60-70 hours per student. That means, we worked about 130 hours more than required.

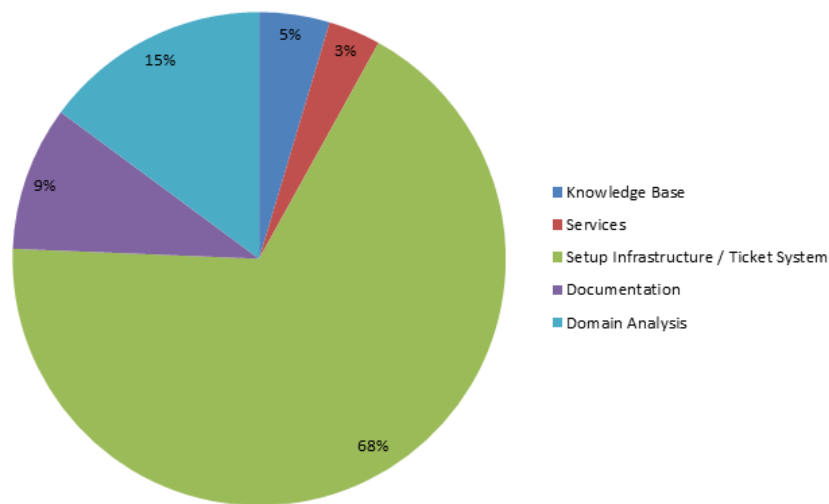
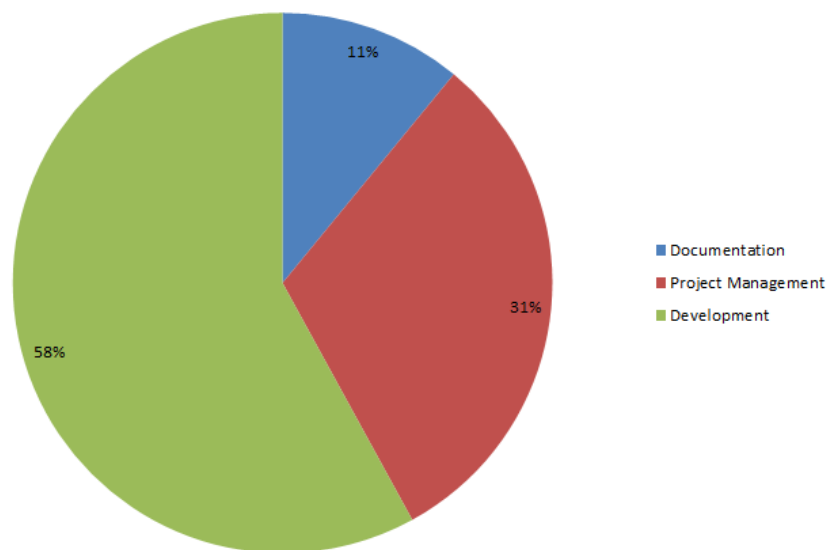


Figure 26: Feature Diagram

The setup of the infrastructure and the Ticket System feature could not be separated because they went hand in hand. This diagram primarily shows the large amount of time that was required to build the system. It shows the fact that the project was oversized from the very beginning.





*Figure 27: Activity Diagram*

Project Management contains the weekly meetings, the setup of the environment but also the analysis and project planning part of the bachelor thesis. Development refers to actual implementation and code testing.

Also, this diagram shows the huge amount of implementation work for the infrastructure.

## 8.2. Division of work

The project was split up between the students. The .NET part is made by Simon Zingg, the Android application is written by Rico Beti. Project management, architectural, conceptual and design-related work have always been done together as a team.

## 8.3. Communication

A common problem in communication is that many things are talked about, but they are never written down. This fact lead us to start recording all decisions starting at the beginning of the project. In order to avoid drowning in the e-mail flood and in order to increase the transparency we have decided to setup a forum in which the decision-making process takes place (see appendix forum).

Furthermore, for more dynamic documents like use cases or diagrams, we used a

shared Google Drive directory. Especially in Skype meetings, this approach was very useful. All participants could post messages or images, which were immediately visible to all members.

The students met every week on two half days to share the ideas and discuss solution approaches. These meetings were indispensable and ensured that everyone was pulling in the same direction. Every second or third week, we had meeting with Frank Koch. We exchanged the system state and clarified open questions. The communication with HELVETAS mostly went via the forum or by e-mail.

#### 8.4. Risk Management

Risk management should be carried out from time to time. Risks are not static, they change over and over again. The following listing is a summary of the risks we experienced while working on this project:

- **Frameworks:** Unknown frameworks can lead to dumb errors, which are very time consuming. To avoid this, we decided to mainly use frameworks we already know. A result of this strategy was the use of Windows Forms, Entity Framework, ASP.NET, NLog and AutoMapper. Despite this measure, we had some trouble with Entity Framework Migration in the beginning.
- **Underrate some requirements:** This is a perennial problem. Our risk management consisted in generous planning. We also wanted to budget writing documentation and team meetings. If not for these attempts, it is almost impossible to make a good planning.
- **Communication problem:** In theory, a project gets harder as the distance between the project member rises. After experiencing this fact, we decided in cooperation with our supervisor Frank Koch only to propose our point of view and our solutions. If we did not receive a helpful response in a reasonable period of time, we went on with our solution. Unfortunately an other approach was not practicable.
- **Problems with Android:** The main issue on Android was the overall poor documentation of libraries across the board combined with general instability. External bugs in third party software (i.e. in Android Support Library and Active Android) caused some delays. Furthermore, the development tools seem to be not as established as other alternative such as Visual Studio. One way to combat these issues would be to regularly update development tools and

dependencies, as well as trying to stick to official Android libraries instead of relying on third party code.

- **Performance issues:** As seen in other projects, especially with Entity Framework and ASP.NET, they quickly turn into a performance nightmare if you are not careful. We decided to always keep an eye on the performance. This approach preserved us from performance problems.
- **Usability:** Most of the projects are developed back end first. We chose the front end first approach, which lead us to designing the GUI first. After the confirmation from HELVETAS, we started with the implementation. Through regular communication with HELVETAS, we have tried to minimize the risk of developing into the wrong direction.

## 9. Documentation

A good documentation is an integral part of this bachelor thesis due to the fact that the HELVETAS Horticulture System will continue to be developed after this project has come to an end.

We have decided to create an extensive code documentation by rigorously commenting the source code so that a documentation can be generated

In addition to the code documentation, we have created several guides (see appendices/guides) that will assist the HELVETAS development team to setup the project and get familiarized with it.

### 9.1. Doxygen

Doxygen allows the creation of code documentations for developers based on specific comments within the source files (i.e. XML Code Tags in C# and JavaDoc in Java).

The manual can be generated or updated by executing the following command on the command line in the root folder of the project repository: `doxygen doxygen-conf`. The generated documentation will be placed in the directory `./doc/html/` relative to the root folder.

Additional information regarding usage and configuration can be found on the official Doxygen website<sup>28</sup>. Dimitri's Doxygen site<sup>29</sup> provides additional helpful information regarding the basic syntax used for documentation comments.

The entire code documentation (i.e. for Server, Windows Client, and Android App) has been attached to this theses and can be found in the appendices.

### 9.2. Server Online API Documentation

ASP.NET offers an online API documentation out of the box. The documentation is generated dynamically at runtime and is also based on documentation comments within the source code.

For convenience, the server API documentation has been attached to this thesis as an offline version that can be found in the appendices.

---

<sup>28</sup><http://www.doxygen.org/>

<sup>29</sup><http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html>

## List of Figures

1	Domain Model .....	13
2	Use Case Diagram .....	20
3	UX: Posts & Threads .....	32
4	UX: Login, Services and News Feed .....	32
5	Test Coverage .....	34
6	Code Metrics .....	36
7	System Overview.....	37
8	Deployment Diagram.....	38
9	Code-First Approach .....	45
10	Server Dependency Diagram .....	47
11	View-ViewModel-Model.....	50
12	SSD UI-Command Binding .....	51
13	Admin Client Dependency Diagram.....	52
14	Android Lifecycle Flow Chart .....	53
15	Android Client Local Database .....	55
16	Android Client Local Database .....	56
17	Database Model .....	58
18	UC Help Query - Communication Diagram .....	62
19	UC Process Ticket - Communication Diagram.....	63
20	ThreadFilter Class Diagram.....	64
21	SSD ThreadFilter .....	64
22	Knowledge Base: Android Search .....	67
23	HAC diagram.....	70
24	HAC algorithm test .....	71
25	Time per Student.....	80
26	Feature Diagram .....	80
27	Activity Diagram .....	81

## References

- [1] “akshay”, *Improving Image Compression – What We’ve Learned from WhatsApp*, <https://www.built.io/blog/2013/03/improving-image-compression-what-weve-learned-from-whatsapp/>, March 4<sup>th</sup> 2014 [Accessed: November 23<sup>rd</sup> 2015]
- [2] “Wiegers, Karl E.”, *Listening to the Customer’s Voice*, <http://www.processimpact.com/articles/usecase.html>, March 1997 (originally) [Accessed: December 1<sup>st</sup> 2015]
- [3] “Dykstra, Tom”, *Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application*, <http://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>, July 30<sup>th</sup> 2013 [Accessed: December 2<sup>nd</sup> 2015]
- [4] “Entity Framework Tutorial”, *Entity Framework Code-First*, <http://www.entityframeworktutorial.net/code-first/entity-framework-code-first.aspx>, 2015 [Accessed: December 9<sup>th</sup> 2015]
- [5] “Semantic Search Art”, *Hierarchical Agglomerative Clustering*, <http://www.semanticquery.com/archive/semanticsearchart/researchHAC.html>, [Accessed: November 24<sup>th</sup> 2015]
- [6] “Schwichtenberg, Holger”, *Sinn und Unsinn der Windows Communication Foundation*, <http://www.heise.de/developer/artikel/Sinn-und-Unsinn-der-Windows-Communication-Foundation-934625.html>, February 22<sup>nd</sup> 2010 [Accessed: November 30<sup>th</sup> 2015]

# Appendices

## A. Attachments

The following documents are attached to this bachelor thesis (in logical order):

- Project Assignment (German) `./Project Assignment.pdf`
- Personal Statements `./Personal Statements.pdf`
- HHS Poster `./HHS Poster.pdf`
  
- HHS Source Code `./source/Helvetas Horticulture System.zip`
  
- HHS Server `./binaries/server/Server.zip`
- HHS Administration Client `./binaries/admin/HHSAdmin.exe`
- HHS Android App `./binaries/androidapp-debug.apk`
  
- HHS Client & Android Doc `./documentation/HHS Client & Android.zip`
- HHS Server API Doc `./documentation/HHS Server API.zip`
  
- API Keys `./appendices/api_keys/*`
- Diagrams & Models `./appendices/diagrams/*`
- Project Forum Backup `./appendices/forum/*`
- Installation Guides `./appendices/guides/*`
- Code Metrics & Time Log `./appendices/misc/*`
- SMS Gateway Workaround `./appendices/smsg/*`
- HHS Load Generator `./appendices/hhsperf.zip`

## B. Licenses

All third party libraries that have been used for the implementation of this project as well as their licenses will be specified consecutively.

### HHS.API

- ViBlend UI Components (free components only)  
<http://www.viblend.com/WinFormsControlsLicenseComparison.aspx>

### HHS.Admin

- ASP.NET (Apache License Version 2.0)
- jQuery (MIT License (compatible with GPL 3.0))  
*Required only for auto-generated Server API Online Documentation.*
- Bootstrap (MIT License)  
*Required only for auto-generated Server API Online Documentation.*
- Simple Injector (MIT License)

### HHS.Common

- NLog (BSD License)
- Castle.Core (Apache License Version 2.0 (compatible with GPL 3.0))

### HHS.BLL

- Entity Framework (Apache License Version 2.0)
- AutoMapper (MIT License)
- Quartz.NET (Apache License Version 2.0)
- Newtonsoft.Json (MIT License)



**Android App**

- Google Cloud Messaging (Apache License Version 2.0)
- Google Location Services (Apache License Version 2.0)
- Retrofit 2.0 (Apache License Version 2.0)
- Retrofit GSON-Converter (Apache License Version 2.0)
- OkHttp Logging Interceptor (Apache License Version 2.0)
- Active Android (Apache License Version 2.0)
- Joda Time (Apache License Version 2.0)